



AT&T

999-802-2141S

System Programmer's Guide

AT&T Personal Computer
6300 PLUS

©1985 AT&T
All Rights Reserved
Printed in USA

NOTICE

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

GW-BASIC is a trademark of Microsoft Corp.

IBM is a registered trademark of International Business Machines Corporation.

MS-DOS and MICROSOFT are registered trademarks of Microsoft Corporation.

UNIX is a registered trademark of AT&T

CONTENTS

1

System Programming Concepts

Purpose of This Guide	1-3
Conventions Used in This Guide	1-4
Terms Used in This Guide.....	1-8
Abbreviations Used in This Guide	1-10
Programming Steps	1-13
Service Information	1-15
Parts Catalog	1-17
General Documentation— AT&T Personal Computer 6300PLUS	1-18
Technical Documentation— AT&T Personal Computer 6300 PLUS	1-19
MS-DOS 3.10 Operating System Documentation	1-20
UNIX System V Release 2.0 Foundation Set Documentation	1-21
UNIX System Software Development Documentation	1-23

2 LINK Program

LINK File Usage	2-4
Segments, Groups, and Classes	2-7
Invoking LINK	2-9
LINK Switches	2-13
Command Line Entry	2-17
Sample Link Session	2-20
LINK Error Messages	2-22

3 DEBUG Program

Overview	3-3
Terms Used in DEBUG	3-4
How to Start DEBUG	3-7
DEBUG Commands	3-9
DEBUG Error Messages	3-57

4

Addressing Scheme

Overview	4-2
Memory Addressing	4-3
Registers	4-8
Flags	4-10
Addressing Modes	4-13

5

Memory and Disk Allocation

Overview	5-3
Memory Configuration	5-4
Memory Map	5-5
Low Memory Map	5-6
ROM BIOS Data Area	5-7
ASCIIZ Strings	5-8
Handles	5-9

File Control Blocks	5-10
Disk Layout	5-16
Disk Directory	5-17
Disk Boot Sector	5-24
File Allocation Table (FAT)	5-27
Disk Formats	5-33

6 Program Structure and Loading

Overview	6-2
Pros And Cons For Selecting a Program File Format	6-3
EXE2BIN Program	6-6
File Header Format	6-9
Program Segment Prefix	6-12
Program Loading Process	6-15
Relocation Process For .EXE Files	6-20

7

System Calls

Overview	7-7
Table of Interrupts	7-8
Numerical Table of Functions	7-9
Alphabetical Table of Functions	7-12
Programming Considerations	7-15
Interrupts (INT)	7-16
INT 20H - Program Terminate	7-17
INT 21H - Function Request	7-19
INT 22H - Terminate Address	7-20
INT 23H - Control-C Exit Address	7-21
INT 24H - Fatal Error Abort Address	7-22
INT 25H - Absolute Disk Read	7-27
INT 26H - Absolute Disk Write	7-30
INT 27H - Terminate but Stay Resident	7-33
Functions	7-35
00H - Terminate Program	7-36
01H - Read Keyboard and Echo	7-38
02H - Display Character	7-40
03H - Auxiliary Input	7-41

04H - Auxiliary Output	7-42
05H - Print Character	7-44
06H - Direct Console I/O	7-46
07H - Direct Console Input	7-49
08H - Read Keyboard	7-51
09H - Display String	7-53
0AH - Buffered Keyboard Input	7-54
0BH - Check Keyboard Status	7-57
0CH - Flush Buffer, Read Keyboard	7-59
0DH - Reset Disk	7-61
0EH - Select Disk	7-63
0FH - Open File	7-64
10H - Close File	7-67
11H - Search for First Entry	7-69
12H - Search for Next Entry	7-72
13H - Delete File	7-74
14H - Sequential Read	7-76
15H - Sequential Write	7-79
16H - Create File	7-82
17H - Rename File	7-84
19H - Get Current Disk	7-86
1AH - Set Disk Transfer Address	7-87
1BH - Get Default Drive Data	7-89
1CH - Get Drive Data	7-91
21H - Random Read	7-93
22H - Random Write	7-96

23H - Get File Size	7-99
24H - Set Relative Record	7-102
25H - Set Interrupt Vector	7-104
26H - Create New PSP	7-106
27H - Random Block Read	7-107
28H - Random Block Write	7-110
29H - Parse File Name	7-113
2AH - Get Date	7-117
2BH - Set Date	7-119
2CH - Get Time	7-121
2DH - Set Time	7-123
2EH - Set/Reset Verify Flag	7-125
2FH - Get Disk Transfer Address	7-127
30H - Get MS-DOS Version Number	7-128
31H - Keep Process	7-129
33H - Control-C Check	7-130
35H - Get Interrupt Vector	7-132
36H - Get Disk Free Space	7-134
38H - Get/Set Country Data	7-136
39H - Create Directory	7-142
3AH - Remove Directory	7-144
3BH - Change the Current Directory	7-146
3CH - Create Handle	7-148
3DH - Open Handle	7-150
3EH - Close Handle	7-152
3FH - Read Handle	7-154

40H - Write Handle	7-156
41H - Delete a Directory Entry	7-158
42H - Move File Pointer	7-160
43H - Get/Set Attributes	7-163
4400H, 4401H - IOCTL Data	7-165
4402H, 4403H - IOCTL Character	7-168
4404H, 4405H - IOCTL Block	7-170
4406H, 4407H - IOCTL Status	7-172
4408H - IOCTL is Changeable	7-174
4409H - IOCTL is Redirected Block	7-176
440AH - IOCTL is Redirected Handle	7-178
440BH - IOCTL Retry	7-180
45H - Duplicate File Handle	7-182
46H - Force Duplicate of a Handle	7-184
47H - Get Current Directory Path	7-186
48H - Allocate Memory	7-188
49H - Free Allocated Memory	7-190
4AH - Set Block	7-192
4B00H - Load and Execute a Program	7-194
4B03H - Load Overlay	7-198
4CH - End Process	7-201
4DH - Get Return Code of Child Process	7-203
4EH - Find First File	7-204
4FH - Find Next File	7-206
54H - Get Verify State	7-208
56H - Change Directory Entry	7-209

57H - Get/Set Date/Time of File	7-211
58H - Get/Set Allocation Strategy	7-214
59H - Get Extended Error	7-217
5AH - Create Temporary File	7-221
5BH - Create New File	7-224
5C00H - Lock	7-226
5C01H - Unlock	7-229
5E00H - Get Machine Name	7-232
5E02H - Printer Setup	7-234
5F02H - Get Assign List Entry	7-236
5F03H - Make Assign List Entry	7-239
5F04H - Cancel Assign List Entry	7-242
62H - Get PSP	7-244
General Macros	7-245

8

BIOS Routines

Overview	8-4
Routines	8-5
Routine Interrupt Table	8-6
INT 5H - Print Screen Routine	8-7
INT 10H - Video Routines	8-8
INT 11H - Equipment Check Routine	8-19
INT 12H - Real Mode Memory Size Routine	8-21

INT 13H - Disk Routines	8-22
INT 14H - Communication Routines	8-27
INT 15H - Protected Mode Memory Size Routine	8-31
INT 16H - Keyboard Routines	8-32
INT 17H - Printer Routines	8-43
INT 19H - Bootstrap Routine	8-45
Bypassing The BIOS Routines	8-47
ROM BIOS Listing.....	8-48

9

Device Drivers

Overview	9-5
Background.....	9-8
Device Headers.....	9-15
Request Header	9-20
Hardware Controller Documentation	9-35
Asynchronous Communications Element (ACE)	9-36

Communications Manager Interface	9-45
Display Controller	9-66
DMA Controller	9-82
Floppy Diskette Controller (FDC)	9-92
Hard Disk Unit Controller	9-118
Keyboard Interface	9-139
Mouse (Optional)	9-144
Parallel Printer Interface	9-147
Programmable Interrupt Controller (PIC)	9-151
Programmable Interval Timer	9-161
Real-Time Clock And Calendar	9-166
Speaker	9-172
Switching Between Real and Protected Modes	9-175

10 Display Enhancement Board

DEB Capabilities	10-3
Programming Tips	10-8
How To Program The DEB.....	10-10
Interrupt 10H Functions	10-13
Programming the LUT	10-30

11 Mouse

Mouse	11-3
List of Mouse Functions	11-7
Mouse Function Calls	11-8

A

Compatibility

Software Compatibility Considerations	A-4
The AT&T PC 6300 PLUS Versus the AT&T PC 6300	A-9
MS-DOS 3.1 Versus MS-DOS 2.X	A-12

G

Glossary

I

Index



Purpose of This Guide	1-3
Conventions Used in This Guide	1-4
Command and Statement Text Syntax	1-4
Type Styles	1-5
Displaying of Prompts and Messages	1-6
Examples	1-6
Terms Used in This Guide.....	1-8
Bit	1-8
Byte	1-8
Chapter	1-8
Double Word	1-9
Extension.....	1-9
Filename	1-9
Name	1-9
Nibble	1-9
Pathname	1-9
Word	1-9
Abbreviations Used in This Guide	1-10

Programming Steps	1-13
Running High-Level Language	1-13
Writing Assembler Programs	1-13
Writing Utilities	1-13
Programming Devices Directly	1-14
Service Information	1-15
Parts Catalog	1-17
General Documentation	1-18
Technical Documentation	1-19
MS-DOS 3.1.0 Operating System Documentation	1-20
UNIX System V Release 2.0 Foundation Set Documentation	1-21
UNIX System V Release 2.0 Software Development Documentation	1-23

Purpose of This Guide

This guide provides you with in-depth information on the AT&T Personal Computer 6300 PLUS program development tools. The guide focuses on what you need to know to use existing AT&T Personal Computer hardware and hardware interfaces.

The final chapter on programming devices assumes that you have a working knowledge of the principles of designing device drivers. This chapter gives you the technical details on how to program the AT&T PC 6300 PLUS.

Conventions Used in This Guide

Throughout this guide, you will notice these conventions used in text to improve clarity.

**COMMAND
AND
STATEMENT
TEXT
SYNTAX**

The following syntax is used in descriptions of command and statement text:

- `[]` Square brackets indicate that the enclosed entry is optional. For example, if the path is optional it will appear in text as `[path]`.
- `{ }` Braces indicate a choice between two or more entries, separated with commas. At least one of the entries enclosed in braces must be chosen. For example, if A, B, and C are the choices, they will appear in text as `{A,B,C}`.
- `...` Ellipses indicate that an entry may be repeated as many times as needed.
- `<>` Angle brackets indicate that the enclosed entry is part of a 1-word phrase. That is, multiword text is broken up with angle brackets for easier reading. For example, the word `gettextendederror` will appear in text as `<get><extended><error>`.

- ☐ Boxes indicate labeled keys. For example, the RETURN key is labeled like this **RETURN** in text.
- CTRL** CTRL followed by a single letter indicates a control code. Control codes are generated by holding down the **CTRL** key and then typing a single letter immediately following **CTRL**. For example, the control S code looks like this **CTRLS** in text.

Note: The space bar is shown as **SPACE** in text, but is not actually labeled “SPACE” on the key.

TYPE STYLES

There are several different type styles used. Four of these styles have special meaning:

- One style, **bold**, is used when illustrating command and variable entries.
- Another style, *italic*, is used when referring to “text files.” Also, this type style is used to emphasize keywords.
- The third style, **monospace**, is used in the various figures and displays. Also, this type style is used in text when referring to messages or prompts seen on your terminal’s screen.
- The fourth style, **monospace bold**, is used when describing required responses.

DISPLAYING OF PROMPTS AND MESSAGES

System-displayed items are centered within the main body of the text so that you can easily distinguish them. For example, LINK prompts:

OBJECT MODULES[.OBJ]

System-displayed items representing a series of inputs and outputs will be set inside an outline. For example:

```
$ pwd
\level1\level2\filename
$
```

EXAMPLES

Discussion in text is sometimes followed by examples. In multistep examples, all steps are numbered, and displays following each step are indented. Current typing will be bold in the display.

Here is a typical example from the DEBUG Chapter:

To display and change the SP register to 0000:

1 Type RSP **RETURN**.

DEBUG displays:

```
-RSP
SP FFF0
```


2 Type 0000 **RETURN**.

The value of the SP register, which was FFF0, is changed to 0000. DEBUG displays:

```
-RSP  
SP FFF0  
-0000
```

Terms Used in This Guide

Throughout this guide, you will notice these popular terms used in text. See the glossary for a more complete list.

BIT Refers to a single unit of data or information. A Bit can indicate one of two states usually indicated by "0" or "1".

Bit Status

The status of a bit is referred to by several terms, all of which indicate "0" or "1". The following table refers to the various terms used to refer to the status of a bit.

0	1
reset	set
no	yes
off	on

BYTE Refers to an 8-bit unit of information. A Byte can indicate 256 different states usually indicated by 00 through FF (hexadecimal).

CHAPTER Refers to a tabbed section in this guide as a chapter in text.

DOUBLE WORD	Refers to a 32-bit or 2-word unit of information, typically indicating a complete address that includes the segment and offset.
EXTENSION	Refers to the extension part of a filename used to identify types of files.
FILENAME	Refers to the entire name including the extension. Filename = name + extension.
NAME	Refers to the name (up to 8 characters) of the file; for example, myprog.
NIBBLE	Refers to a 4-bit unit of information. A Nibble can indicate 16 different states usually indicated by 0 through 9, then A through F.
PATHNAME	[d:][path]name[.ext] Refers to the entire file entry that contains the drive number, path, name, and extension; for example, B:\sys\mydir\myprog.exe
WORD	Refers to a 16-bit or 2-byte unit of information, typically indicating an address such as a segment or offset.

Abbreviations Used in This Guide

Throughout this guide, you will notice these abbreviations used in text.

ACE	Abbreviation for asynchronous communications element.
ADRS	Abbreviation for address.
APA	Abbreviation for all points addressable.
BIOS	Abbreviation for basic input/output system.
BPB	Abbreviation for BIOS parameter block.
CNTS	Abbreviation for contents.
ECC	Abbreviation for error correcting code.
EOF	Abbreviation for end-of-file.
EOI	Abbreviation for end of interrupt.
FAT	Abbreviation for file allocation table.

FCB	Abbreviation for file control block.
H	Symbol placed after a number to refer to its value in Hexadecimal. "H" is used in text when it is not immediately obvious that the number is in Hexadecimal.
ID	Abbreviation for identification.
INT	Abbreviation for interrupt. Used extensively in Chapter 7 to distinguish interrupts from functions.
I/O	Input/Output: Common reference used in text when discussing input and/or output devices.
K	Symbol placed after a number to refer to the value "thousand" when discussing size in bytes, but actual value is 1024. For example, if memory is listed at 64K, that is referred to as "64 thousand" but is actually 65,536 bytes.
LSB, MSB	Least significant byte, most significant byte: The terms LSB and MSB are used to indicate which byte of a 2-byte word is being discussed. The LSB is bits 0 through 7 of the word and the MSB is bits 8 through 15.

M	Symbol placed after a number to refer to the value “million” when discussing size in bytes (2^{20}), actual value is 1,048,576 or 1024K bytes.
NPX	Abbreviation for numeric processor extension.
N/A	Not applicable: Common reference used to indicate that the item does not apply in this particular situation.
PSP	Abbreviation for program segment prefix.
RAM	Abbreviation for random access memory.
ROM	Abbreviation for read only memory.
UART	Abbreviation for Universal Asynchronous Receiver/Transmitter.

This is not a complete list. Some abbreviations used in text are defined at that point.

Programming Steps

This section shows where to go for information on the task you are performing.

RUNNING HIGH- LEVEL LANGUAGE

To run a program via the GWBASIC interpreter, see **Chapter 7, SYSTEM CALLS**. You can call these functions via a GWBASIC program.

To run a compiled program, read **Chapter 2, LINK**, as well as **Chapter 7**.

WRITING ASSEMBLER PROGRAMS

Chapters 1 through 8 are aimed at programmers writing assembler programs. If you have not used the 8086 assembly language, **Chapter 4, ADDRESSING SCHEME**, gives you a good start. **Chapters 2 and 3, LINK and DEBUG**, respectively, are fundamental to writing and debugging assembler programs. Also, read **INTERRUPTS and FUNCTION CALLS** in **Chapter 7**.

WRITING UTILITIES

To write a supplementary utility program, read:

- Sections on "Writing Assembler Programs" (listed above)
- **Chapters 5 and 6—MEMORY MAPS, CONTROL BLOCKS, AND DISKETTE ALLOCATION, and PROGRAM STRUCTURE AND LOADING**, respectively.

**PROGRAM-
MING
DEVICES
DIRECTLY**

Every section applies to writing device drivers, especially **Chapter 9, DEVICE DRIVERS**.

Service Information

In the event that you have a problem with your AT&T Personal Computer 6300 PLUS and want to have it repaired, contact the AT&T Information Systems Service Organization by calling 1-800-922-0354 (unless special instructions have been otherwise provided to you). The person who answers your call will provide specific instructions depending on your individual needs and maintenance contract options.

AT&T Information Systems offers five Equipment Maintenance Agreement Plans:

1. **Business Day Service:** Contracted; 5-day (M-F) 8 a.m. to 5 p.m., Systems Technician dispatched to customer location, if required, for equipment repair. Time charges apply outside coverage period.
2. **Around-the-Clock Service:** Contracted; 24 hours a day, 7 days a week on major failures with Systems Technician dispatched to customer location, if required, for equipment repair.
3. **Dedicated Service:** Technician dedicated to specific location in 1-, 2-, or 3-shift coverage, for 5-, 6-, or 7-day week.

4. **Per Occurrence:** Noncontracted; technician dispatched to customer location, charged on time-and-material basis. Time only is charged for contracted customers who want service out of contracted hours.
5. **Mail-In:** Replacement part(s) will be mailed to the customer. Upon receipt of the replacement part(s), the customer repackages and returns the defective part(s) to AT&T in accordance with the instructions provided by the AT&T Information Systems Services Hotline (1-800-922-0354).

Parts Catalog

AT&T Information Systems offers an AT&T Personal Computer 6300 PLUS parts catalog that can be ordered, at no charge, by placing a toll-free call to the National Parts Sales Center (1-800-222-PART). This catalog details the major customer replaceable modules of the AT&T Personal Computer 6300 PLUS. It also tells you how to order any documentation that you may need.

General Documentation— AT&T Personal Computer 6300 PLUS

Getting Started With Your

AT&T Personal Computer 6300 PLUS

This guide introduces the major parts of the PC 6300 PLUS, shows how to load the MS-DOS and UNIX Operating Systems, and describes tests that help to isolate any problems with the computer.

Installation Guide

AT&T Personal Computer 6300 PLUS

This guide contains procedures for unpacking and installing the PC 6300 PLUS. Instructions are provided for connecting a printer to the PC 6300 PLUS.

Reference Cards

The PC 6300 PLUS General Quick Reference Cards briefly describe the PC 6300 PLUS Keyboard and general operating procedures.

Technical Documentation— AT&T Personal Computer 6300 PLUS

Service Manual

AT&T Personal Computer 6300 PLUS

This document supports field service technicians in the installation, diagnostic, and maintenance of the PC 6300 PLUS. Appendixes to this manual discuss DIP switch and jumper settings in addition to information on field replaceable modules.

Hardware Reference Manual

AT&T Personal Computer 6300 PLUS

This manual is a complete reference guide to the hardware of the PC 6300 PLUS. Major components and options available for the PC 6300 PLUS are described as well as their interfaces. Detailed information is provided on the internal architecture, buses, and components of the system unit.

System Programmer's Guide

AT&T Personal Computer 6300 PLUS

This document provides in-depth information on the PC 6300 PLUS program development tools that allow a sophisticated programmer to write application programs.

MS-DOS 3.10 Operating System Documentation

MS-DOS by Microsoft® User's Guide

This guide introduces the basic concepts of MS-DOS, explains the most commonly used MS-DOS commands, and provides an alphabetical listing of each of these MS-DOS commands.

GW BASIC by Microsoft® Programmer's Guide

This guide is a reference for programmers wishing to use the GW Basic programming language.

MS-DOS Reference Cards

The MS-DOS Quick Reference Cards briefly describe the most commonly used MS-DOS 3.10 procedures and commands.

UNIX System V Release 2.0 Foundation Set Documentation

UNIX System V Release 2.0 Operations Guide AT&T Personal Computer 6300 PLUS

This guide describes the UNIX System shell and other UNIX System features used to administer, configure, and maintain the UNIX System on the PC 6300 PLUS. Procedures are given for running an MS-DOS application from the UNIX System and switching from the MS-DOS screen to the UNIX screen with the touch of a key.

UNIX System V Release 2.0 User Reference Manual System Maintenance Manual AT&T Personal Computer 6300 PLUS

This manual describes the features of the UNIX System and is divided into two subsections: a User Manual and a System Maintenance Manual. The User Manual sections cover system commands and application programs, file formats, and miscellaneous facilities. The System Maintenance Manual contains system maintenance programs and special files.

UNIX System V Release 2.0 User Guide

This document contains an introduction to the UNIX System and tutorials for the UNIX text editors and communication facilities.

Simul-Task Software Guide for MS-DOS Applications

This guide is to be used as an aid for installing, using, and removing MS-DOS application programs under Simul-Task OS Merge. "Application Notes" are included for some of the most popular MS-DOS applications.

UNIX System Reference Cards

The PC 6300 PLUS UNIX System Quick Reference Cards briefly describe the most commonly used UNIX System procedures and commands.

Simul-Task OS Merge Reference Cards

The PC 6300 PLUS Simul-Task OS Merge Quick Reference Cards briefly describe the most commonly used Simul-Task OS Merge procedures and commands.

Office Reference Cards

The PC 6300 PLUS Office Quick Reference Cards briefly describe how to use the Office, including the Office menu and other menus associated with the Office.

UNIX System Software Development Documentation

UNIX System V Release 2.0 Software Development Guide

AT&T Personal Computer 6300 PLUS

This guide describes the installation and use of the Software Development Package for the PC 6300 PLUS. It provides a guide for developing software to take advantage of the special user interface and MS-DOS compatibility features offered by the PC 6300 PLUS.

UNIX System V Release 2.0 Programmers Reference Manual

AT&T Personal Computer 6300 PLUS

This manual describes in detail the system calls, libraries, subroutines, and file formats of the UNIX System on the PC 6300 PLUS.

UNIX System V Release 2.0 Support Tools Guide

This guide describes the various software "tools" that are available to aid the UNIX System user.

UNIX System V Release 2.0 Programming Guide

This guide describes the two main programming languages (C Language and Fortran) supported on the UNIX System.



Overview	2-3
LINK File Usage	2-4
Syntax	2-4
VM.TMP File	2-5
Changing Diskettes	2-6
Segments, Groups, and Classes	2-7
Segment	2-7
Group	2-7
Class	2-8
Invoking LINK	2-9
Ways To Invoke LINK	2-9
LINK Prompts	2-10
Object Modules To Be Included	2-11
LINK Switches	2-13
/DSALLOCATE	2-13
/HIGH	2-14
/LINENUMBERS	2-14
/MAP	2-14
/PAUSE	2-15
/STACK: <number>	2-15
/NO	2-16

Command Line Entry	2-17
Syntax	2-17
Automatic Response File Entry	2-18
Sample Link Session	2-20
LINK Error Messages	2-22

Overview

LINK is an executable program on your MS-DOS Supplemental Programs diskette. LINK combines object modules that are the output of the MACRO-86 assembler or compatible compiler. It produces a relocatable run file (load module) and a list file of external references and error messages.

To run LINK, you provide object, run, list, and library file parameters. You may optionally enter switches that modify the operation of LINK.

"Invoking the Linker" describes the three ways to run LINK: interactive entry, command line entry, and automatic response file entry. Interactive entry is used most frequently, so its section contains information common to all three methods.

When you link a high-level language program, the compiler determines the arrangement of your object modules in memory. When you use assembler, however, you have more control over your program's organization. The section "Segments, Groups, and Classes" shows you how to specify the order of your object modules at run time.

LINK File Usage

The LINK process involves the use of several files.

LINK:

- Works with one or more input files
- Produces two output files
- Creates a temporary disk file if necessary
- Searches up to eight library files.

The format for LINK file specifications is the same as that of any disk file:

SYNTAX **[d:][pathname]filename**

d: The drive designation. Permissible drive designations for LINK are A: through O: .

pathname A path of directory names.

filename Any legal filename of one to eight characters, plus an optional extension of 1 to 3 characters.

ext

A 1- to 3-character extension to the filename.

If no filename extensions are given in the input (object) file specifications, LINK recognizes the following extensions by default:

.OBJ Object
.LIB Library.

LINK appends the following default extensions to the output (Run and List) files:

.EXE Run (may not be overridden)
.MAP List (may be overridden).

**VM.TMP
FILE**

LINK uses available memory for the link session. If an output file exceeds available memory, LINK creates a temporary file, names it VM.TMP, and puts it on the disk in the default drive. If LINK creates VM.TMP, it will display the message:

VM.TMP has been created.
Do not change diskette in drive, <d:>

Once this message is displayed, do not remove the diskette from the default drive until the link session ends. If the diskette is removed, the operation of LINK is unpredictable and LINK usually displays the error message:

Unexpected end of file on VM.TMP

LINK writes the contents of VM.TMP to the file named following the Run File: prompt. VM.TMP is a working file only and is deleted at the end of the linking session.

Do not use VM.TMP as a filename for any file. If LINK requires the VM.TMP file, LINK deletes the VM.TMP already on disk and creates a new VM.TMP. Thus, the contents of the previous VM.TMP file are lost.

CHANGING DISKETTES

You may want to change diskettes during the link operation. If LINK cannot find an object file on the specified diskette, it prompts you to change diskettes rather than aborting the session. When you enter the /PAUSE switch, LINK pauses and prompts you to change diskettes before it creates the run file. You may change diskettes when prompted except in the following cases:

- The diskette you want to change has a VM.TMP file on it.
- You have requested a list file on the diskette you want to change.

Segments, Groups, and Classes

The following terms are explained to help you understand how LINK works. Generally, if you are linking object modules from a high-level language compiler, you do not need to know these terms. If you are linking assembly language modules, read this section carefully.

SEGMENT

A segment is a contiguous area of memory up to 64K bytes in length. A segment may be located anywhere in 8086 memory on a "paragraph" (16 bytes) boundary. The contents of a segment are addressed by a segment-register:offset pair.

GROUP

A group is a collection of segments. When you use assembly language, you name the group. For high-level languages, the compiler names the group.

The group is used for addressing segments in memory. Each group is addressed by a single segment register. The segments within the group are addressed by the segment register plus an offset. LINK checks to see that the total sizes of the object modules within one group do not exceed 64K bytes. A program consists of one or more groups.

Assembler programmers can benefit from using groups. Grouping helps with referring to data items within related segments, without worrying about using segment overrides or changing segment registers.

CLASS

A class is a collection of segments. You assign the class name for assembly language programs; the compiler assigns the class name for high-level language programs.

Segments are grouped together by the class name found in the object modules. Classes may be loaded across 64K bytes byte boundaries. LINK divides classes into groups that are no longer than 64K bytes.

Example:

Segment Name	Name	Segment Class
Segment 1	PROG.1	CODE
Segment 2	PROG.2	CODE
Segment 12	PROG.3	DATA

Note that segments 1, 2, and 12 have different segment names, but may or may not have the same segment class name. Segments 1, 2, and 12 form a group with a group address of the lowest address of segment 1 (i.e., the lowest address in memory).

Each segment has a segment name and a class name. LINK loads all segments into memory by class name from the first segment encountered to the last. All segments assigned to the same class are loaded into memory continuously.

During processing, LINK references segments by their address in memory (where they are located). LINK does this by finding groups of segments.

Invoking LINK

WAYS TO INVOKE LINK

LINK is invoked in one of three ways. The first method, interactive entry, requires you to respond to individual prompts.

For the second method, command line entry, type all commands on the same line used to start LINK.

To use the third method, automatic response file entry, create a response file that contains all the necessary commands and tell LINK where that file is when you run LINK.

Interactive Entry	LINK
Command Line Entry	LINK filenames[/switches]
Automatic Response File Entry	LINK @filename

Interactive Entry

To invoke LINK interactively, type:

LINK

LINK loads into memory, then displays four prompts, one at a time. At the end of each line, you may type one or more switches preceded by a forward slash (/).

The command prompts are summarized below. Defaults appear in square brackets ([]) after the prompt. Object Modules is the only prompt that requires a response.

LINK PROMPTS

Object Modules[.OBJ]:	[d:][path]name[.ext] [+d:][path][name].ext]...
Run File[name.EXT]:	[d:][path][name[.ext]]
List File[NUL.MAP]:	[d:][path][name[.ext]]
Libraries[.LIB]:	[d:][path][name[.ext]] [+d:][path]name[.ext]]...

Note 1: When you enter a filename without specifying the drive, the default drive is assumed. When you enter a filename without specifying the path, the default path is assumed. The libraries prompt is an exception—if the linker looks for the libraries on the default drive and doesn't find them, it looks on the drive specified by the compiler.

Note 2: To select default responses to the remaining prompts, use a single semicolon (;) followed immediately by **RETURN** at any time after the first prompt (Run File:).

Once you enter the semicolon, you can no longer respond to any of the prompts for that link session. Use **RETURN** to skip prompts.

Note 3: Use **CTRL**C to abort the link session at any time.

**OBJECT
MODULES
TO BE
INCLUDED****Object Modules [.OBJ]:**

List .OBJ files to be linked. They must be separated by blank spaces or plus signs (+). If the plus sign is the last character typed, this prompt will reappear so that you can enter more object modules.

LINK assumes that object modules have the extension .OBJ unless you explicitly specify some other extension. Object filenames may not begin with the @ symbol (@ is used for specifying an automatic response file).

The order in which you key in the object files is significant. See the section on segments, groups, and classes for more information.

Load Module**Run File [Obj-file.EXE]:**

Give filename for executable object code. The default is <first-object-filename>.EXE. You cannot change the output extension. You can specify just the drive designation or just a path.

Listing**List File [NUL.MAP]:**

Give filename for listing (also known as a linker map). The listing is not created if you select the default. You can request a listing by entering a drive designator, path, or name[.ext]. If you do not specify an extension, the default .MAP is used.

You can have the listing printed by specifying a print device instead of a filename or have the listing displayed on the screen by specifying CON. If you display the linker map, you can also print it by pressing **CTRL** and **PrtSc**.

Libraries To Be Searched

Libraries [.LIB]:

List filenames to be searched, separated by blank spaces or plus signs (+). If a plus sign is the last character typed, the prompt will reappear.

LINK searches library files in the order listed to resolve external references. When it finds the module that defines the external symbol, LINK processes that module as another object module.

There is no default library search for MACRO assembler object modules. For compiled modules, if you select the default for this prompt, LINK looks for the compiler package's library on the default drive. If not found there, LINK looks on the drive specified by the compiler.

If LINK cannot find a library file, it displays:

Cannot find library <library-name>
Type new drive letter:

Press the letter for the drive designation (for example, B). If two libraries have the same filename, only the first in the list is searched.

LINK Switches

The seven LINK switches control various LINK functions. Type switches at the end of a prompt response regardless of which method you use to start LINK. Switches may be grouped at the end of any response, or may be scattered at the end of several. If you type more than one switch at the end of one response, each switch must be preceded by a forward slash (/).

All switches may be abbreviated. The only restriction is that an abbreviation must be sequential from the first letter through the last typed, no gaps or transpositions are allowed. For example the following is a list of legal and illegal abbreviations for the /DSALLOCATE switch:

Legal	Illegal
/D	/DSL
/DS	/DAL
/DSA	/DLC
/DSALLOCA	/DSALLOCT

/DSALLOCATE /DSALLOCATE tells LINK to load all data at the high end of the data segment. Otherwise, LINK loads all data at the low end of the data segment. At run time, the DS pointer is set to the lowest possible address to allow the entire DS segment to be used.

Use of /DSALLOCATE in combination with the default load low (that is, the /HIGH switch is not used) permits the user application to dynamically allocate any available memory below the area specifically allocated within DGroup yet to remain addressable by the same DS pointer. This dynamic allocation is needed for Pascal and FORTRAN programs.

Your application program may dynamically allocate up to 64K bytes (or the actual amount of memory available) less the amount allocated with DGroup.

/HIGH

/HIGH causes LINK to place the Run file as high as possible in memory. Otherwise, LINK places the Run file as low as possible.

Note: Do not use /HIGH with Pascal or FORTRAN programs.

/LINENUMBERS

/LINENUMBERS tells LINK to include in the List file the line numbers and addresses of the source statements in the input modules. Otherwise, line numbers are not included in the List file.

Not all compilers produce object modules that contain line number information. In these cases, of course, LINK cannot include line numbers.

/MAP

/MAP directs LINK to list all public (global) symbols defined in the input modules. If /MAP is not given, LINK will list only errors (including undefined globals).

The symbols are listed alphabetically. For each symbol, LINK lists its value and its segment:offset location in the Run file. The symbols are listed at the end of the List file.

/PAUSE

/PAUSE causes LINK to pause in the link session when the switch is encountered. Normally, LINK performs the linking session from beginning to end without stopping. This switch enables you to swap the diskettes before LINK outputs the Run (.EXE) file.

When LINK encounters /PAUSE, it displays the message:

About to generate .EXE file
Change disks <hit any key>

LINK resumes processing when you press any key.

Note: Do not remove the disk which will receive the List file, or the disk used for the VM.TMP file, if one has been created.

**/STACK:
<number>**

Stack number represents any positive numeric value (in hexadecimal radix) up to 65,536 bytes. If a value from 1 to 511 is typed, LINK will use 512. If /STACK is not used for a link session, LINK calculates the necessary stack size automatically.

All compilers and assemblers should provide information in the object modules that allow the linker to compute the required stack size.

At least one object (input) module must contain a stack allocation statement. If not, LINK will display the following error message:

WARNING:NO STACK STATEMENT

/NO

/NO is short for NO DEFAULT LIBRARY-SEARCH. This switch applies only to higher level language modules. This switch tells LINK not to search the default libraries in the object modules.

For example, when you are linking object modules in Pascal, specifying /NO tells LINK not to automatically search the library named PASCAL.LIB to resolve external references.

Command Line Entry

You may invoke LINK by typing all commands on one line. The entries following LINK are responses to the command prompts. The entry fields for the different prompts must be separated by commas. Use the following syntax:

SYNTAX **LINK**<obj-list>,<runfile>,<listfile>,<lib-list>[/switch...]

obj-list A list of object modules, separated by plus signs

runfile Name of the file to receive the executable output

listfile Name of the file to receive the listing

lib-list List of library modules to be searched

/switch Optional switches that may be placed following any of the response entries (just before any of the commas or after the <lib-list>, as shown).

To select the default for a field, simply type a second comma with no spaces between the two commas.

Example:

```
LINK
FUN+TEXT+TABLE+CARE, ,FUNLIST,
COBLIB.LIB
```

This command causes LINK to load, then the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ are loaded. LINK links the object modules and writes the output to FUN.EXE (by default), creates a List file named FUNLIST.MAP, and searches the library file COBLIB.LIB.

AUTOMATIC RESPONSE FILE ENTRY

It is often convenient to save responses to the linker for reuse at a later time. This is especially useful when a long list of object modules needs to be specified.

Before using this option, you must create the response file. Each line of text corresponds to one LINK prompt. The responses must be typed in the same order as they are when entered interactively. To continue a line, type a plus sign (+) at the end of the line.

You can enter the name of more than one automatic response file on the command line and combine response file names with additional parameters. The combined series of resulting parameters must be a valid sequence of LINK prompts.

Use switches and special characters (+ and ;) in the response file the same way they are used when entered interactively.

To invoke the linker using a response file, type:

LINK @ <filename>

Filename is the name of a response file. The extension is not required.

When the session begins, LINK displays each prompt with the corresponding response from the response file. If the response file does not contain answers for all the prompts, LINK displays the prompt that does not have a response and waits for a response. If you type a legal response, LINK continues the link session.

Example:

```
FUN TEXT TABLE CARE
/PAUSE/MAP
FUNLIST
COBLIB.LIB
```

This response file tells LINK to load the four object modules named FUN, TEXT, TABLE, and CARE. LINK pauses before producing a public symbol map to permit you to swap disks. When you press any key, the output files will be named FUN.EXE and FUNLIST.MAP. LINK will search the library file COBLIB.LIB and will use the default settings for the switches.

Sample Link Session

This sample shows you the type of information displayed during an LINK session.

In response to the MS-DOS prompt, type:

LINK

The system displays the following messages and prompts:

```
Microsoft Object Linker V2.01
© Copyright 1982, 1983 by Microsoft Inc.

Object Modules [.OBJ]: IO SYSINIT
Run File [IO.EXE]:
List File [NUL.MAP]: IO /MAP
Libraries [.LIB_];
```

Note 1: By specifying /MAP, you get both an alphabetic listing and a chronological listing of public symbols.

Note 2: By responding PRN to the List File: prompt, you can redirect your output to the printer.

Note 3: By specifying the /LINE switch, LINK gives you a listing of all line numbers for all modules. (Note that /LINE can generate a large volume of output.)

Once LINK locates all libraries, the linker map displays a list of segments in the order of their appearance within the load module. The list might look like this:

Start	Stop	Length	Name
00000H	009ECH	09EDH	CODE
009F0H	01166H	0777H	SYSINITSEG

The information in the Start and Stop columns shows the 20-bit hexadecimal address of each segment relative to location zero. Location zero is the beginning of the load module.

The addresses displayed are not the absolute addresses where these segments are loaded. See Chapter 3 (DEBUG Program) for information on how to determine the absolute address of a segment.

Because the /MAP switch was used, LINK displays the public symbols by name and value. For example:

ADDRESS	PUBLICS BY NAME
009F:0012	BUFFERS
009F:0005	CURRENT MS-DOS LOCATION
009F:0011	DEFAULT DRIVE
009F:000B	DEVICE LIST
009F:0013	FILES
009F:0009	FINAL MS-DOS LOCATION
009F:000F	MEMORY SIZE
009F:0000	SYSINIT

ADDRESS	PUBLICS BY VALUE
009F:0000	SYSINIT
009F:0005	CURRENT MS-DOS LOCATION
009F:0009	FINAL MS-DOS LOCATION
009F:000B	DEVICE LIST
009F:000F	MEMORY SIZE
009F:0011	DEFAULT DRIVE
009F:0012	BUFFERS
009F:0013	FILES

LINK Error Messages

All errors cause the link session to abort. After the cause has been found and corrected, LINK must be rerun. The following error messages are displayed by LINK:

**Attempt to access data outside of segment bounds,
possibly bad object module**

There is probably a bad object file.

Bad numeric parameter

Numeric value is not in digits.

Cannot open temporary file

LINK is unable to create the file VM.TMP because the disk directory is full. Insert a new disk. Do not remove the disk that will receive the LIST.MAP file.

Error: dup record too complex

DUP record in assembly language module is too complex. Simplify DUP record in assembly language program.

Error: fixup offset exceeds file width

An assembly language instruction refers to an address with a short instruction instead of a long instruction. Edit assembly language source and reassemble.

Input file read error

There is probably a bad object file.

Invalid object module

An object module(s) is incorrectly formed or incomplete (as when assembly is stopped in the middle).

Symbol defined more than once

LINK found two or more modules that define a single symbol name.

Program size or number of segments exceeds capacity of linker

The total size may not exceed 384K bytes and the number of segments may not exceed 255.

Requested stack size exceeds 64K bytes

Specify a size greater than or equal to 64K bytes with the /STACK switch.

Segment size exceeds 64K bytes

64K bytes is the addressing system limit.

Symbol table capacity exceeded

Many and/or very long names were typed exceeding the limit of approximately 50K bytes.

Too many external symbols in one module

The limit is 256 external symbols per module.

Too many groups

The limit is 10 groups.

Too many libraries specified

The limit is 8 libraries.

Too many public symbols

The limit is 1024 public symbols.

Too many segments or classes

The limit is 256 (segments and classes together total 256).

Unresolved externals: <list>

The external symbols listed have no defining module among the modules or library files specified.

VM read error

This is a disk error; it is not caused by LINK.

Warning: no stack segment

None of the object modules specified contains a statement allocating stack space, but you typed the /STACK switch.

Warning: segment of absolute or unknown type

There is a bad object module or an attempt has been made to link modules that LINK cannot handle (e.g., an absolute object module).

Write error in TMP file

No more disk space remains to expand the VM.TMP file.

Write error on run file

Usually, there is not enough disk space for the Run file.

Overview	3-3
Terms Used in DEBUG	3-4
Address	3-4
Drive	3-5
Keyletter	3-5
Parameter	3-5
Range	3-5
Value	3-6
How to Start DEBUG.....	3-7
Examples	3-8
DEBUG Commands	3-9
A (ASSEMBLE)	3-12
C (COMPARE)	3-16
D (DISPLAY)	3-18
E (ENTER)	3-20
F (FILL).....	3-24
G (GO).....	3-26
H (HEX ARITHMETIC)	3-29

I (INPUT)	3-31
L (LOAD)	3-32
M (MOVE)	3-35
N (NAME)	3-37
O (OUTPUT)	3-40
P (PROCEED)	3-41
Q (QUIT)	3-42
R (REGISTER)	3-44
S (SEARCH)	3-48
T (TRACE)	3-50
U (UNASSEMBLE)	3-52
W (WRITE)	3-54
DEBUG Error Messages	3-57

Overview

DEBUG is an executable object program that resides on your MS-DOS Supplemental Programs diskette. DEBUG performs the following functions:

- Steps through a program, instruction by instruction, to examine registers and memory locations for testing purposes.
- Changes register and file contents during the DEBUG session so you can test a code change without reassembling your program.
- Makes permanent changes to disk files so you can use DEBUG to save files that may otherwise be lost.
- Translates machine code instructions into assembly language equivalents for testing purposes, with the U (Unassemble) command.

Note: DEBUG only debugs 8086 microprocessor code. The PC 6300 PLUS process in the real mode is compatible with the 8086 microprocessor.

Terms Used in DEBUG

First, let's define the following DEBUG terms that are used in text:

ADDRESS

Refers to a specific memory location. There are several methods of addressing which allow you to point to specific memory locations, either directly or via a segment register.

An address consists of a segment value and an offset value in this format:

`segment:offset`

If the colon separator is missing, DEBUG assumes the value is an offset and will use either the DS or CS register for the segment value, depending upon the command used.

A segment register may be specified instead of an actual segment value like this:

`ES:0090`

All of these are valid address parameters:

0200
A03B
09AC:000A
3A02:0A11
DS:0300
SS:A004

DRIVE Refers to a specific drive selection either 0, 1, or 2 depending on whether you wish to select drive A, drive B or drive C, respectively. For example, to select drive B type:

1:

KEYLETTER Refers to the single-letter designation (A,C,D,...) for the DEBUG command.

PARAMETER Refers to the additional information following the keyletter that specifies exactly what the command is to perform — mostly used to override defaults. Some commands do not require parameters and some parameters are optional.

RANGE Refers to a range of addresses (memory locations). There are several formats which can be used to specify the range.

When you know the start and end addresses, use this format:

<start address>,<end address>

When you know the start address and how many locations (value) are in the range, use this format:

<start address>L<value>

Remember that **value** must be indicated in hexadecimal. For example, to address a range of 50 locations starting with 0100, type:

0100L32

In both formats the range represents an offset. Therefore, the specified range cannot be greater than FFFF.

You can also specify a segment address before the range like this:

DS:0300L32

VALUE

Refers to the contents or data of a memory address in a 1- to 4-character hexadecimal value.

How to Start DEBUG

Start the DEBUG program by typing:

SYNTAX **DEBUG [filename][,arglist]**

filename The name of the file to be debugged.

arglist An optional list of filename parameters and switches, separated by commas. These will be passed to the program specified by the filename parameter.

When the program is loaded into memory, it is loaded as if you had loaded the file without the DEBUG prefix.

When you enter DEBUG without a filename, you can manipulate current memory disk blocks, or disk files since no filename has been specified.

On entering the DEBUG environment, DEBUG responds with the hyphen "-" (prompt).

You now may enter any DEBUG command.

The DEBUG session will continue until you type **Q** or **q** to quit.

COMMENTS When you include the filename in the command line, the specified file is loaded into memory starting at address 100H.

EXE Files When you specify a file with a .EXE extension, the program is loaded into memory starting at the address specified in the header of the file.

HEX Files When you specify a file with a .HEX extension, the program is loaded into memory starting at the address specified in the HEX file. HEX files are in INTEL hexadecimal format and are converted to memory image format by DEBUG.

EXAMPLES To begin a DEBUG session without loading a file, type:

DEBUG **RETURN**

To begin a DEBUG session with the "myprog" file loaded into memory from drive B, type:

DEBUG b:myprog **RETURN**

Starting DEBUG sets up a program segment prefix at offset 0 in the program work area. You can overwrite this area by entering DEBUG without parameters.

Moreover, when you are debugging a file with a COM or EXE extension, do not tamper with the program header below location 5CH, or DEBUG will terminate.

Do not attempt to run a program after a **Program terminated normally** message is displayed. You must reload the program with the N (Name) and L (Load) commands for it to run properly.

DEBUG Commands

The following commands are for performing the DEBUG program functions:

A - ASSEMBLE	Assembles 8086 microprocessor mnemonics directly into memory.
C - COMPARE	Compares the contents of two areas of memory.
D - DISPLAY	Displays an area of memory.
E - ENTER	Changes the value of consecutive memory addresses.
F - FILL	Fills a range of memory addresses with specified values.
G - GO	Executes the program currently in memory, optionally pausing at specified addresses to display information about the system and program environment.
H - HEX	Calculates and displays the sum and the difference of two hexadecimal values.
I - INPUT	Inputs a value from an address (I/O port) and displays the value.
L - LOAD	Loads a file or absolute disk blocks into memory.

M - MOVE	Copies (moves) the values of a specified range of memory addresses (source) to another range of memory addresses (destination).
N - NAME	Provides filenames for the L (Load) and W (Write) commands or filename parameters for the program to be debugged.
O - OUTPUT	Outputs a value to an address (I/O port) and displays the value.
P - PROCEED	Executes one or more instructions and displays the register values, flag settings, and the next instruction to be executed. Proceed treats interrupts, subroutine calls, repeat-string instructions, and loop instructions as a single operation.
Q - QUIT	Terminates the DEBUG program.
R - REGISTER	Displays the contents of the registers and flag settings, with the option to change the values.
S - SEARCH	Searches a range of addresses for a specified value(s).
T - TRACE	Executes one or more instructions and displays the register values, flag settings and the next instruction to be executed.
U - UNASSEMBLE	Disassembles strings of values in memory and displays them as assembler-like statements along with their corresponding addresses.
W - WRITE	Writes a file or absolute disk blocks to disk from memory.

The following rules apply to the use of DEBUG commands:

- Commands can be entered in either lowercase or uppercase.
- All commands are entered using the keyletter followed by the parameters. Here are some examples:

```
D0100,03A0
c09ACLFF,0300
MCS:100,110,CS:500
```

Command parameters can be separated from each other by spaces or commas for readability but need not be, except where two hexadecimal numbers are entered as parameters, in which case they must be separated by a comma or space. For brevity, the syntax of this chapter will always indicate a comma where separation is obligatory, but note that a space can alternatively be used.

- Commands become effective only after typing **RETURN**.
- If you make a syntax error when entering a command, the **Error** message will be displayed on the following line. You must re-enter the command using the correct syntax.

Any DEBUG command may be aborted at any time by pressing **CTRL**C. Pressing **CTRL**S suspends the display, so that you can read it before the output scrolls away. After suspending the display, press any key (except **CTRL**S or **CTRL**C) to continue scrolling.

The DEBUG commands follow individually in alphabetical order.

A (ASSEMBLE)

Assembles 8086 microprocessor mnemonics directly into memory.

SYNTAX **A[address]**

address Address is the start address into which the subsequently entered line of mnemonics is to be assembled. If this parameter is omitted, location 100 is assumed, if you did not enter an Assemble command previously. If you did enter Assemble, the code assembles into the address following the last instruction loaded by the previous Assemble command.

After you enter the Assemble command, DEBUG displays the specified address followed by the cursor. You may then enter a line of 8086 microprocessor mnemonics. On terminating the line with **RETURN**, the line will be assembled into memory starting at the specified location. The subsequent address to the assembled code will be displayed on the next line along with the cursor to enable you to enter the next line of code. If, instead of a line of 8086 microprocessor mnemonics, you simply type **RETURN**, the Assemble command terminates and the DEBUG prompt reappears.

COMMENTS

- All values are hexadecimal and must be entered as 1 to 4 characters. Prefix mnemonics must be specified in front of the opcode to which they refer. You may also enter them on a separate line.

- The segment override mnemonics are CS, DS, ES, and SS. The mnemonics for the far return is RETF. String manipulation mnemonics must explicitly state the string size. For example, use MOVSB to move byte strings.
- The Assemble command will automatically assemble short, near, or far jumps and calls, depending on byte displacement with respect to the destination address. These may be overridden with the **NEAR** or **FAR** prefix. For example:

```
0100:0500 JMP 502           ;a two-byte short jump
0100:0502 JMP NEAR 505      ;a three-byte near jump
0100:0505 JMP FAR 50A       ;a five-byte far jump
```

- The NEAR prefix may be abbreviated to NE, but the FAR prefix cannot be abbreviated.
- DEBUG cannot tell whether some operands refer to a word memory location or to a byte memory location. In this case the data type must be explicitly stated with the prefix **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. For example:

```
NEG BYTE PTR [128]
DEC WO [SI]
```

- DEBUG cannot distinguish whether an operand refers to a memory location or to an immediate operand. Enclose operands that refer to memory locations in square brackets. For example:

```
MOV AX,21           ;Load AX with 21H
MOV AX,[21]         ;Load AX with the contents of
                    ;location 21H
```

- Two pseudo-instructions are available with the Assemble command. The DB opcode will assemble word values directly into memory. For example:

```
DB 1,2,3,4,"THIS IS AN EXAMPLE"  
DB "THIS IS A QUOTE:"  
DB "THIS IS A QUOTE'"  
DW 1000,2000,3000,"BACH"
```

- The Assemble command supports all forms of register indirect addressing. For example:

```
ADD BX,34[BP+2]. [SI-1]  
POP [BP+DI]  
PUSH [SI]
```

- All opcode synonyms are supported. For example:

```
LOOPZ 100  
LOOPE 100  
JA 200  
JNBE 200
```


EXAMPLE

To assemble starting at location 200:

- 1 Type A200 **RETURN**.

DEBUG displays:

```
-A200  
09AC:200
```

- 2 Type MOV AX,[21] **RETURN**.

The 8086 microprocessor mnemonics are assembled starting at location 200. The next address subsequent to the assembled code is then displayed:

```
-A200  
09AC:0200MOV AX,[21]  
09AC:0203
```

- 3 Type **RETURN**.

The Assemble command terminates and the DEBUG prompt reappears.

```
-A200  
09AC:0200MOV AX,[21]  
09AC:0203
```

C (COMPARE)

Compares the contents of two areas of memory.

SYNTAX **C<range>,address**

range The range of addresses defining the first area to be compared. If no segment is specified, then the entire segment specified in the DS register is assumed.

address The first address of the area to be compared with the area specified by the range parameter.

The Compare command compares the area of memory specified by the range parameter with an area of the same size starting at the address specified by the address parameter.

COMMENTS

- If the contents of the two areas are identical, nothing is displayed.
- If there are differences, then the differences are displayed (per-line) in the format:

adrs1 cnts1 cnts2 adrs2

where **adrs1** indicates the address in the first area and **cnts1** its contents. And where **adrs2** indicates the corresponding address in the second area and **cnts2** its contents.

EXAMPLE

To compare 100 addresses starting at A500 with 100 addresses starting at B200:

- 1 Type CA500,A564,B200 **RETURN**, or
CA500L64,B200 **RETURN**.

DEBUG displays:

```
-CA500L64,B200
A502 00A4 00A3 B202
A503 0001 0000 B203
A530 09AA 034C B230
```

After comparing addresses (locations) A500 through A564 with addresses B200 through B264, the display indicates that addresses A502, A503, and A530 were different.

D (DISPLAY)

Displays an area of memory.

SYNTAX **D[range]**
 or
 D[address]

range The range of addresses whose contents are to be displayed. If you enter only an offset, the segment specified in the DS register is assumed.

address The address from which the display is to start. The contents of this address and the next 127 locations are displayed.

COMMENTS

- If **D** is specified without parameters, then the 128 addresses following the last address to be accessed are displayed. If no address has yet been accessed, the display will start from location DS:100.
- If **D** and the range parameter are specified, the contents of that range of addresses are displayed. If this takes more than 24 screen lines, the display is scrolled until the contents of the final address in the range are displayed on line 24.
- The address values are displayed (per-line) in this format with a hyphen "-" between **v+7** and **v+8**:

adrs v v+1 ... v+15 <ASCv><ASCv+1>...<ASCv+15>

where **adrs** indicates the address, **v** its hexadecimal value, **v+1** the hexadecimal value of the next address, and so on. **ASCv** through **ASCv+15** indicate the ASCII characters of the corresponding address. The character dot "." is displayed when the value does not have a printable ASCII character.

Each line displays the hexadecimal value and ASCII character for 16 addresses. The next line starts with the address following the **v+15** location.

EXAMPLE 1 To display the values of addresses 0100 through 0110 type:

1 Type D100,110 **RETURN**.

DEBUG displays the address followed by the next 16 addresses, with a hyphen "-" between addresses 8 and 9. Then the ASCII character of all 16 addresses, if printable.

```
-D0100,0110
09AC:0100 E8 09 ... 9B-DF ... 01 .....
09AC:0110 00
```

EXAMPLE 2 To display the next 128 addresses:

1 Type D **RETURN**.

DEBUG displays:

```
-D
09AC:0110 54 48 ... 20-54 ... 20 THIS IS TEXT IN
09AC:0120 44 41 ... 43-41 ... 46 DATA LOCATIONS F
09AC:0130 4F 52 ... 4D-50 ... 54 OR A SAMPLE OF T
09AC:0140 48 45 ... 4C-41 ... 41 HE DISPLAY COMMA
09AC:0150 4E 44 ... 48-45 ... 20 ND IN THE DEBUG
09AC:0160 50 52 ... 00-00 ... 00 PROGRAM.....
09AC:0170 00 00 ... 00-00 ... 00 .....
09AC:0180 00 00 ... 00-00 ... 00 .....
```

E (ENTER)

Changes the value of consecutive memory addresses.

SYNTAX **E**<address>[,value,value+1,...]

address The address of the location whose value is to be changed, or the address of the first of a succession of locations whose contents are to be replaced. If only an offset is specified, then the segment indicated by the DS register is assumed.

value The data that is to replace the contents of the specified address. The contents of the memory locations are changed. That is, the first **value** will replace the contents of the location specified by the address. A second **value** will replace the contents of the location following that specified by the **address**, and so on.

COMMENTS

- If the command is entered without value parameters, DEBUG displays the specified address and its value. This command then waits for you to perform one of the following:
 - 1 Change the displayed value by entering another value. To do this, you type the new value after the current value. If you enter an illegal value, or if you type more than two digits, the illegal or extra character is not echoed.
 - 2 Type **[SPACE]** to continue to the next address. To change this value, simply enter a new value as described above. To

advance to the next address without changing the current value, press **SPACE** again.

- 3 Type **□**, (hyphen) to return to the previous address. DEBUG then starts a new display line with the address you have returned to and its value. You can change the value of this address as described above. To move back one address further without changing this value, type **□** again, and another new line will be displayed. You can return to the address specified in the command.

- When you advance beyond eight addresses on one line, DEBUG starts a new display line with the address displayed at the start of the line.
- When you specify **values**, the first of these values will replace the value at the address specified by the address parameter. Subsequent values in the command will replace subsequent values of addresses in memory.

EXAMPLE 1 To change the value of three consecutive addresses starting at address 100:

- 1 Type E0100 **RETURN**.

DEBUG displays:

```
-E0100
058D:0100 CD.
```

- 2 Type 26.

The value of address 100, which was CD, is now changed to 26. DEBUG displays:

```
-E0100
058D:0100 CD.26
```

3 Type **SPACE**.

DEBUG displays the value of the next address (101):

```
-E0100  
058D:0100 CD.26 20.
```

4 Type 19.

The value of address 101, which was 20, is now changed to 19. DEBUG displays:

```
-E0100  
058D:0100 CD.26 20.19
```

5 Type **SPACE**.

DEBUG displays the value of the next address (102):

```
-E0100  
058D:0100 CD.26 20.19 0A.
```

6 Type 23 **RETURN**.

The value of address 102, which was 0A, is now changed to 23. The enter command is terminated and DEBUG displays:

```
-E0100  
058D:0100 CD.26 20.19 0A.23
```


EXAMPLE 2 Another way to change the value of the same three addresses:

1 Type E0100,26,19,23 **RETURN**.

The values of addresses 100, 101, and 102 which were CD, 20, and 0A are now changed to 26, 19, and 23. DEBUG displays:

```
-E0100,26,19,23
058D:0100 CD.26
058D:0101 20.19
058D:0102 0A.23
```

EXAMPLE 3 If you change the first and third addresses and not the second address (that is, you typed **SPACE** instead of changing the second address) the display would look like this:

```
-E0100
058D:0100 CD.26 20. 0A.23
```

EXAMPLE 4 If you change the first two addresses, but change your mind on the value in the second address (that is, you typed **□** instead of a new value for the third address) the display would look like this:

```
-E0100
058D:0100 CD.26 20.19 0A.
058D:0101 19.
```

F (FILL)

Fills a range of memory addresses with specified values.

SYNTAX **F<range>,value[,value+1,value+2...]**

range The range of addresses whose values are to be changed with the specified values. If only the offset is specified, then the segment indicated by the DS register is assumed.

value A 2-digit hexadecimal value that is to change the value of the specified address(es).

COMMENTS

- If the specified range contains more addresses than the list of values, then the list of values is repeated until the specified range is filled. If a range of six addresses and three values are specified, the first and fourth, second and fifth, and third and sixth addresses will contain the same values.
- If the list of values contains more values than the range has addresses, the extra values are ignored.

EXAMPLE

To fill 100 addresses with 00 starting at address 300:

1 Type F0300L64,00 **RETURN**.

DEBUG displays:

-F0300L64,00

The display indicates that addresses 0300 through 0364 (hexadecimal for 100) now contain 00.

G (GO)

Executes the program currently in memory, optionally pausing at specified addresses to display information about the system and program environment.

SYNTAX **G[=address][,address...]**

=address The address in memory at which program execution is to start. "=" is used to distinguish a start address from a pause address.

address The pause address or addresses. You can specify up to ten addresses, in any order.

When you enter **G** without parameters, the program currently in memory is executed starting from the address specified by the CS and IP registers. The program will not pause.

COMMENTS

- When you specify the =address parameter, the contents of the CS and IP registers are changed to those specified by the =address parameter and the program in memory is executed, starting from the address you specified.
- When you specify one or more pause addresses, program execution stops at the first such address encountered and displays the contents of the registers, the state of the flags and the next instruction to be executed (see the R [register] command for a description of the display).

- If only an offset is entered for an address, the command assumes the segment in the CS register.
- When you enter more than ten addresses, DEBUG will display:

BP Error

- Before executing the program, the command changes the value of the pause addresses (locations) with an interrupt instruction (CC). Then program execution will pause at these addresses.

DEBUG restores the original values of all the specified addresses. However, if the program terminates normally (that is, not at a specified address), the original values are not restored.

Note: Once a program has reached completion (DEBUG has displayed **Program terminated normally**), you must reload the program before you can run it again.

- Each pause address that you specify must point to the first address of an 8086 microprocessor instruction or unpredictable results occur.
- The stack segment (SS register) must have six addresses available for this command; otherwise, unpredictable results occur.

EXAMPLE

To run a program starting at address 300 with pauses at addresses 303, and 308 (assume program extends beyond address 308):

1 Type G=0300,0303,0308 RETURN.

DEBUG displays:

-G=0300,0303,0308

```
AX=0000 BD=0000 CS=0000 DX=0000
SP=FFFF BP=0000 SI=0000 DI=0000 DS=0000
ES=0000 CS=0000 IP=013B
NV UP EI PL NZ NA PO NC
0000:013B 083D8      MOV DS,AX
```

```
AX=0000 BD=0000 CS=0000 DX=0000
SP=FFFF BP=0000 SI=0000 DI=0000 DS=0000
ES=0000 CS=0000 IP=013B
NV UP EI PL NZ NA PO NC
0000:013B 083D8      MOV DS,AX
Program terminates normally
```

H (HEX ARITHMETIC)

Calculates and displays the sum and the difference of two hexadecimal values.

SYNTAX **H<value-a>,<value-b>**

value-a The first of two hexadecimal values.

value-b The hexadecimal value that is to be added to or subtracted from value-a.

COMMENTS

- The hexadecimal values may be up to four characters long.
- The HEX command displays two 4-digit values; the first is the result of adding value-b to value-a and the second is the result of subtracting value-b from value-a.

EXAMPLE

To determine the sum and the difference of
019F and 010A:

1 Type H019F,010A **RETURN**.

DEBUG displays:

```
-H019F,010A  
02A9 0095
```

To determine the sum and the difference of
FFFF and 0002:

1 Type HFFFF,0002 **RETURN**.

DEBUG displays:

```
-HFFFF,0002  
0001 FFFD
```


I (INPUT)

Inputs a value from an address (I/O port) and displays the value.

SYNTAX **I<address>**

address The address of the port from which the value is input.

COMMENTS The address, of the I/O port, can be up to 16 bits wide.

EXAMPLE To input a value from address (I/O port) 32F8 (assume address 32F8 is an input port):

1 Type I32F8 **RETURN**.

DEBUG displays:

```
-132F8
0000:32F8 6A
```

L (LOAD)

Loads a file or absolute disk blocks into memory.

SYNTAX **L[address][,drive,block,value]**

address	The starting address in memory at which the file or specified block is to be loaded from. If only an offset is entered, then the segment indicated by the CS register is assumed.
drive	The drive from which disk blocks are to be loaded. For drive A, you must enter 0; for drive B, you must enter 1; for drive C you must enter 2, and so on.
block	The first of a range of blocks to be loaded from the disk specified by the drive parameter.
value	The number of blocks to be loaded.

COMMENTS

- If all parameters are specified, DEBUG loads blocks of information from disk into memory.
- If you specify **L** with just the address parameter, the file whose file control block is correctly formatted at location CS:5C is loaded into memory. The file control block at CS:5C is set either to the filename specified when the DEBUG command was invoked, or to the filename specified by the most recent execution of the N (Name) command.

- If **L** is specified without parameters, the file is loaded starting at address CS:100.
- If **L** and the address parameter are specified, the file is loaded starting at the specified address. In either case, DEBUG sets the BX:CX registers to the number of bytes loaded.
- If the file has an EXE extension, then it is relocated to the load address specified in the header of the EXE file. That is, the address parameter to the Load command is ignored. The header itself is stripped off the EXE file before the file is loaded into memory. The size of the EXE file on disk will differ from its size in memory.
- If the file has an HEX extension, entering the command with no parameters causes the file to be loaded starting at the address specified within the HEX file. However, if the address parameter is specified, then loading starts at the address which is the sum of the address specified and the address in the HEX file.

EXAMPLE 1 To load the file myprog starting at address 0200:

1 Type Nmyprog **RETURN**.

DEBUG displays:

-Nmyprog

The N (Name) command identifies the filename for the load command.

2 Type L0200 **RETURN**.

The file myprog is loaded into memory starting at address 0200. DEBUG displays:

```
-Nmyprog  
-L0200
```

EXAMPLE 2 To relocate the file in memory starting at address 0A00:

1 Type L0A00 **RETURN**.

DEBUG displays:

```
-L0A00
```

M (MOVE)

Copies (moves) the values of a specified range of memory addresses (source) to another range of memory addresses (destination).

SYNTAX **M<range>,address**

range The area of memory whose contents are to be moved. When you only enter an offset, the segment indicated in the DS register is assumed.

address The start of the destination area. When you only entered an offset, the segment indicated by the DS register is assumed.

COMMENTS

- If the source and destination areas overlap, the move is performed without loss of data.

The contents of the source area are not changed by the move, unless the destination area overlaps it.

- When you specify an address as the end of the range, you must only enter the offset. The segment specified, or defaulted to, in the start address of the range is assumed.

EXAMPLE

To copy the values of 32 addresses starting at address 0300 to the area of memory starting at address 0500:

1 Type M0300L20,0500 **RETURN**.

DEBUG displays:

-M0300L20,0500

The 32 addresses specified are copied to memory starting at address 0500.

N (NAME)

Provides filenames for the L (Load) and W (Write) commands or filename parameters for the program to be debugged.

SYNTAX N<filename>[,filename...]

filename The file to be loaded into memory, written to diskette, or used as a filename parameter of the file currently in memory.

The Name command can be used to provide:

- The name of the disk file to be loaded into memory by a subsequent L (Load) command. You can enter the DEBUG program without specifying parameters. You can use this N(NAME)command to name the disk file you wish to debug, and use the L command to load the file into memory. This has the same effect as entering the filename as the first parameter to the DEBUG command. The file control block for the file to be debugged is set up at location CS:5C and the file is loaded.
- The name to be assigned to the file currently in memory when the file is subsequently written to disk. That is, the file is already in memory and the Name command sets up the file control block for the specified file at location CS:5C.

When a W (Write) command is subsequently entered, the file in memory is written to disk with the filename whose file control block is set up at location CS:5C.

- Filename parameters to the file in memory. The file control block, set at CS:5C for the file in memory, is replaced by the first parameter specified. If a second file parameter is specified, the file control block is set up at location CS:6C. Only two file control blocks are set up, although additional filename parameters may be included if required.

All the filenames specified are placed in a save area at CS:81, with CS:80 containing a character count. Parameters specified in this way are analogous to filenames specified in the argument list to the DEBUG command.

COMMENTS None

EXAMPLE 1 To specify the file myprog:

1 Type Nmyprog **RETURN**.

DEBUG displays:

-Nmyprog

The L (Load) and W (Write) commands can now be used.

EXAMPLE 2 To load and run the file myprog starting at address 0300:

1 Type Nmyprog **RETURN**.

DEBUG displays:

-Nmyprog

2 Type L0300 **RETURN**.

The file myprog is loaded into memory starting at address 0300. DEBUG displays:

```
-Nmyprog  
-L0300
```

3 Type G=0300 **RETURN**.

DEBUG runs the file myprog starting at address 0300, and displays:

```
-Nmyprog  
-L0300  
-G=0300
```

O (OUTPUT)

Outputs a value to an address (I/O port) and displays the value.

SYNTAX **O<address>,value**

address The address of the port that the value is to be sent to.

value The contents to be sent, via the specified address, a 2-digit hexadecimal value to the specified port.

COMMENTS The address, of the I/O port, can be up to 16 bits wide.

EXAMPLE To output the value 3C to address (I/O port) 12F8 (assume address 12F8 is an output port):

1 Type O12F8,3C **RETURN**.

DEBUG displays:

```
-012F8,3C
0000:12F8 3C
```

P (PROCEED)

Executes one or more instructions and displays the register values, flag settings, and the next instruction to be executed. Proceed treats interrupts, subroutine calls, repeat-string-instruction and loop instructions as a single operation.

SYNTAX **P[=address],value]**

=address The starting address of partial execution.

value The number of instructions to be executed.

COMMENTS

- If the =address parameter is not specified, execution begins at CS:IP.
- If the value parameter is not specified, only one instruction is executed.
- The registers and flags are displayed in the same format as that of the R (Register) command (without parameters).
- This command is extremely useful in tracing across system calls and procedures.

Q (QUIT)

Terminates the DEBUG program.

SYNTAX Q

The Quit command terminates the DEBUG program without saving the file you are working on. Control is returned to MS-DOS command mode.

COMMENTS None

EXAMPLE To save the file myprog on disk and quit the DEBUG program:

1 Type Nmyprog **RETURN**.

DEBUG displays:

```
-Nmyprog
```

2 Type W **RETURN**.

The file myprog is saved on disk. DEBUG displays:

```
-Nmyprog  
-W
```

3 Type Q RETURN.

The DEBUG program is terminated and displays:

```
-Nmyprog  
-W  
-Q
```

R (REGISTER)

Displays the contents of the registers and flag settings, with the option to change the values.

SYNTAX **R[name][F]**

name Any valid register name whose values are to be examined and/or optionally changed. Valid register names for the 8086 microprocessor are:

AX	DX	SI	ES	IP
BX	SP	DI	SS	PC
CX	BP	DS	CS	

Note: IP and PC both refer to the Instruction Pointer.

F The flag settings are to be displayed and optionally changed. The flags and their values are:

Flag	Set	Clear
Overflow	OV (yes)	NV (no)
Direction	DN (decrement)	UP (increment)
Interrupt	EL (enabled)	DI (disabled)
Sign	NG (negative)	PL (plus)
Zero	ZR (yes)	NZ (no)
Auxiliary	AC (yes)	NA (no)
Parity	PE (even)	PO (odd)
Carry	CY (yes)	NC (no)

COMMENTS

- When you specify **R** without parameters, the values of all registers are displayed along with the flag settings and the next instruction to be executed. For example:

```
-R
AX=058D BD=0000 CS=0000 DX=0000
SP=FFF0 BP=0000 SI=0000 DI=0000 DS=058D
ES=058D CS=058D IP=013B
NV UP EI PL NZ NA PO NC
058D:013B 083D8      MOV DS,AX
```

- When you specify **R** with a register name, DEBUG displays the values of that register. The command then waits for you to do one of the following:
 - 1 Press **RETURN** to terminate the Register command without changing the value of the displayed register.
 - 2 Change the value of the register by entering the two-digit hexadecimal value, terminate the Register command by entering **RETURN**.
- When you specify **RF**, the current flag settings are displayed. You then do one of the following:
 - 1 Press **RETURN** to terminate the Register command without changing the flag values.
 - 2 Change the setting of one or more flags by entering the alternate value of the appropriate flags. The new values may be entered in any order, with or without delimiters.

EXAMPLE 1 To display all registers and flags without changing any values:

1 Type R **RETURN**.

DEBUG displays:

```
-R
AX=058D BD=0000 CS=0000 DX=0000
SP=FFF0 BP=0000 SI=0000 DI=0000 DS=058D
ES=058D CS=058D IP=013B
NV UP EI PL NZ NA PO NC
058D:013B 083D8      MOV DS,AX
```

2 Type **RETURN**.

This terminates the command without changing values. DEBUG displays:

```
-R
AX=058D BD=0000 CS=0000 DX=0000
SP=FFF0 BP=0000 SI=0000 DI=0000 DS=058D
ES=058D CS=058D IP=013B
NV UP EI PL NZ NA PO NC
058D:013B 083D8      MOV DS,AX
```

EXAMPLE 2 To display and change the SP register to 0000:

1 Type RSP **RETURN**.

DEBUG displays:

```
-RSP
SP FFF0
```


2 Type 0000 **RETURN**.

The value of the SP register, which was FFF0, is changed to 0000. DEBUG displays:

```
-RSP
SP FFF0
-0000
```

EXAMPLE 3 To display the flag settings only:

1 Type RF **RETURN**.

DEBUG displays:

```
-RF
NV UP EI PL NZ NA PO NC
```

S (SEARCH)

Searches a range of addresses for a specified value(s).

SYNTAX **S<range>,value[,value+1,value+2,...]**

range The range of addresses within which the search is to be made. If you enter only the offset, the segment indicated by the DS register is assumed.

value The value to be searched for. When searching for more than one value at a time, the values must appear in memory in the same order to be a correct match. In this case, the DEBUG program indicates the address of the first value in the pattern.

For each occurrence of the value(s) within the specified range, DEBUG returns the address of the first value.

COMMENTS If no address is returned, no match was found.

EXAMPLE

To search 256 addresses starting at 0400, for the value 2A:

1 Type S0400L100,2A **RETURN**.

DEBUG displays:

```
-S0400L100,2A
058D:0408
058D:042C
058D:04CA
```

The display indicates that, in the range 0400 through 0500, addresses 0408, 042C, and 04CA contain the value 2A.

T (TRACE)

Executes one or more instructions and displays the register values, flag settings, and the next instruction to be executed.

SYNTAX **T[=address][,value]**

=address The starting address of partial execution.

value The number of instructions to be executed.

COMMENTS

- If the =address parameter is not specified, execution begins at CS:IP.
- If the value parameter is not specified, only one instruction is executed.
- The registers and flags are displayed in the same format as that of the R (Register) command (without parameters).

EXAMPLE

To execute four consecutive instructions in memory starting at address 0200:

1 Type T=0200,4 **RETURN**.

DEBUG executes the instructions associated with addresses 0200 through 0203 and displays:

```
-T=0200,4
AX=058D BD=0000 CS=0000 DX=0000
SP=FFFF BP=0000 SI=0000 DI=0000 DS=058D
ES=058D CS=058D IP=013B
NV UP EI PL NZ NA PO NC
058D:013B 083D8      MOV DS,AX
```

U (UNASSEMBLE)

Disassembles strings of values in memory and displays them as assembler-like statements along with their corresponding addresses.

SYNTAX **U[range]**
 or
 U[address]

range The range of addresses whose values are to be disassembled. If you do not specify the segment, then the segment indicated by the CS register is assumed.

address The address of the value which is to be disassembled. If you do not specify the segment, then the segment indicated by the CS register is assumed.

COMMENTS

- When you specify **U** without parameters, then 32 addresses are disassembled starting at the address specified by the CS:IP register.
- The number of addresses disassembled may be slightly more than the number you specified. This is because instructions are not always the same length and the final address in a range will not always contain the last address of an instruction.

- The first address of a range, or the address parameter, must refer to the first address of an 8086 microprocessor instruction; otherwise, results are unpredictable.

EXAMPLE To disassemble instructions in addresses 0204 through 020B:

1 Type U0204L8 **RETURN**.

DEBUG displays:

```
-U0204L8
058D:0204      8D16DF0D      LEA      DX[0DDF]
058D:0208      42          INC      DX
058D:0209      03D0      ADD      DX,AX
058D:020B      8916E50B      MOV      [0BE5],DX
```

The display indicates what instructions are located in addresses 0204 through 020E.

W (WRITE)

Writes a file or absolute disk blocks to disk from memory.

SYNTAX **W[address][,drive,block,value]**

address The start address of the code in memory that is to be written to disk. If you enter only an offset, then the segment indicated in the CS register is assumed.

drive The drive containing the specified blocks to which code in memory is to be written. For drive A you must enter 0; for drive B you must enter 1; for drive C you must enter 2, and so on.

block The block number on disk that is the first of a contiguous range of blocks to be overwritten with code from memory.

value The number of disk blocks to be overwritten with code from memory.

COMMENTS

- When you specify the command without parameters, the file is written to disk starting from memory address CS:100. When you specify the address parameter, the file in memory, starting from the specified address, is written to disk.

- Before executing the command, BX:CX must be set to the number of bytes to be written. This value was set up correctly when the file was loaded (either by the L [Load] command or the DEBUG command itself). However, if, since loading the file, you have executed a G (Go) or T (Trace) command, then the value of BX:CX has been changed. Be sure this value is set up correctly.
- When the command writes a file to disk, it obtains the drive specifier and filename via the file control block set up at CS:5C. If no drive specifier is set up, then the default is assumed. This file control block is set up either by the DEBUG command (for the file you specify as a parameter to DEBUG) or by subsequent N (Name) command.

If it does not indicate the file specifier you require, you must set up this file control block using the N (Name) command. Refer to "Memory Maps, Control Blocks, and Diskette Allocation" for further details.

- When the file is written to disk, it overwrites the version currently on disk unless the specified filename does not exist, in which case a new file is created.
- If all parameters are specified, then the code in memory is written to the drive specified by the parameter. The data to be written starts at the memory location specified by the address parameter, and is written to the blocks on the disk specified by the **block** and **value** parameters. Be extremely careful to correctly specify the value, since information stored there previously will be destroyed by this operation.

EXAMPLE

To write the file myprog to disk starting from location 0200:

Type Nmyprog **RETURN**.

DEBUG displays:

-Nmyprog

The N (Name) command identifies the filename for the write command.

1 Type W0200 **RETURN**.

The file myprog is written to disk from memory starting at address 0200. DEBUG displays:

-Nmyprog

-W0200

DEBUG Error Messages

BF**Bad Flag**

You attempted **to alter** a flag, but entered some characters that are not acceptable pairs of flag values. See R (Register) command for the list of acceptable flag entries.

BP**Too many Breakpoints**

You specified more than ten breakpoints as parameters to the GO command. Reenter the command with ten or fewer breakpoints.

BR**Bad Register**

You entered the R command with an invalid register name.

DF**Double Flag**

You entered two values for one flag.



Overview	4-2
Memory Addressing	4-3
Real-Address Mode	4-3
Protected-Virtual-Address Mode	4-3
Segmented Addressing	4-4
Pages	4-5
Aligned and Non-Aligned Words	4-7
Registers	4-8
General Registers	4-8
Segment Registers	4-9
Status and Control Registers	4-9
Flags	4-10
Status Flags	4-10
Control Flags	4-12
Addressing Modes	4-13
Register and Immediate Addressing	4-14
Memory Addressing	4-14

Overview

The 80286 microprocessor has an extremely flexible addressing scheme. The 80286 microprocessor uses a 16-bit word, but the AT&T Personal Computer 6300 PLUS extends the addressing capability by using two types of addressing: *real* (basic) which can address just over one megabyte, and *virtual* (extended) which addresses over 16 megabytes of memory. Also, the 80286 microprocessor supports eight different addressing modes: two register and immediate addressing modes plus six memory addressing modes.

To take advantage of the flexibility of the 80286 microprocessor, so that you can write assembly language code and step through programs while debugging, study the addressing scheme by carefully reading this chapter.

Memory Addressing

The PC 6300 PLUS utilizes the full addressable space available in the design of the 80286 microprocessor. The basic (real) memory address is 20 bits wide; therefore, the available addressable area is two to the twentieth power (1024 K), or just over one megabyte of memory. The extended (protected-virtual) memory address is 24 bits wide, the available addressable area is sixteen megabytes of memory.

REAL- ADDRESS MODE

The 80286 executes a fully upward-compatible superset of the 8086 instruction set in the real-address mode (real mode). In the real-address mode, the CPU's object code is compatible with the 8086 software.

PROTECTED- VIRTUAL- ADDRESS MODE

The 80286 executes a fully upward-compatible superset of the 8086 instruction set in the protected-virtual-address mode (protected mode). The protected mode also provides memory management and protection mechanisms and associated instructions.

The 80286 enters the protected mode from the real mode by setting the PE (Protection Enable) bit of the machine status word with the Load Machine Status (LMSW) instruction. The protected mode offers extended physical and virtual memory address space, memory protection mechanisms, and new operations to support operating systems and virtual memory.

In the protected mode, the 80286 provides a 1-gigabyte virtual address space per task mapped into a 16 megabyte physical address space. The virtual address space may be larger than the physical address space since any use of an address that does not map to a physical memory location will cause a restartable exception.

SEGMENTED ADDRESSING

The 80286 is a 16-bit microprocessor capable of addressing 16-bit wide memory. The PC 6300 PLUS uses "Segmented Addressing" to accomplish 20-bit (real) or 24-bit (protected-virtual) wide memory addressing. Two 16-bit addresses are converted into one 20-bit or 24-bit address to obtain the actual address.

The two memory addresses are known individually as the **segment** and the **offset**. In text they will appear as:

segment:offset

What actual address in memory will be selected is determined by adding the **offset** to the **segment** once the segment is placed in the upper 16 bits of the 20-bit memory address.

Example

Using real addressing, let's assume you want to find the actual address in memory given:

009F:0012

This specifies a segment of 009F plus an offset of 0012. At this point it is easier if we talk in terms of “nibbles” rather than “bytes”. The data would look like this:

	Nibble			
	4	3	2	1
Segment	0	0	9	F
Offset	0	0	1	2

If we then move the **segment** into the upper 16 bits of the memory address, that is, one “nibble” to the left you can add the **offset** to determine the actual address.

Segment	0	0	9	F	0	
+ Offset			0	0	1	2
<hr/>						
= Actual	0	0	A	0	2	

Therefore, the actual address of 009F:0012 is 00A02.

PAGES

If you think of a page of memory as the area of memory you can address without specifying a new segment, then you can apply the “page” concept to memory addressing for the PC 6300 PLUS.

“Pages” in the 80286 refer to the partial amount of memory that can be accessed without specifying a new segment. Therefore, the 80286 considers a page a 1-segment wide area of memory.

Given that the offset is 16 bits wide, the segment's actual address range is 64K bytes. That is, you can address 64K bytes of memory without changing the segment.

Your program will execute the fastest when instructions do not contain new segment information. Since a given address can appear in several pages, select a segment that starts your program with the least amount of offset. This allows you the greatest amount of progression before a new segment must be specified.

Example

Let's assume a program starts in 0A5C:0007. All of the following point to this address:

0A5C:0007
0734:3287
02A6:7B67

Add an offset of FFFF to each segment above to find the highest location in memory each can address.

Segment	0A5C0
+ Offset	FFFF

= Highest	1A5BF

Likewise, the other segments mentioned have maximums of:

Segment	Highest
0734H	17334H
02A6H	12A5FH

Therefore, the address 0A5C:0007 offers the greatest program size allowed without changing the segment.

ALIGNED AND NON- ALIGNED WORDS

The microprocessor instructions consist of one to six bytes, depending upon the particular instruction. Therefore, instructions will start at either an odd or even address location.

The 80286 microprocessor accesses two bytes (1 word) of data in each memory cycle. An "Aligned" word is accessed by the CPU when the first byte is from an even numbered address, versus the "Non-aligned" word which is accessed by the CPU when the first byte is from an odd numbered address. Non-aligned words require two memory cycles to acquire a complete word, since the 2-byte reading sequence was interrupted by a previous instruction that contained an odd number of bytes.

The importance of aligned or non-aligned words is determined by the execution speed of certain instructions. It is good programming to store data starting at an even address. For programs that access or manipulate many word quantities, this will help speed program execution. When you are writing a device driver and instruction cycle times affect the execution of your program, the impact of aligned and non-aligned words should be taken into consideration.

Registers

The 80286 microprocessor has 14 registers grouped into three main categories: General Registers, Segment Registers, and Status and Control Registers. Each register is 16 bits wide, although some can be used as 8-bit registers.

GENERAL REGISTERS

There are eight general registers which are primarily used for data involved in arithmetic and logical operations. The general registers are labeled AX, BX, CX, DX, SP, BP, SI, and DI.

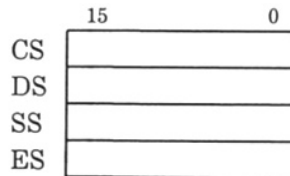
The AX, BX, CX, and DX registers can be used as 8-bit registers by identifying them as the high(H) or low(L) half of the particular register. That is, the AX register can be used as two 8-bit registers by the labels AH and AL. The SP and BP registers are generally used for offsets for stack manipulations. The SI (source) and DI (destination) registers are called index registers because they usually contain an address which is incremented to step through sequential data.

	15	8	7	0
AX	AH		AL	
BX	BH		BL	
CX	CH		CL	
DX	DH		DL	
SP				
BP				
SI				
DI				

**SEGMENT
REGISTERS**

There are four segment registers which contain current addressing segments. These registers are used in combination with other registers or instructions to form the 20-bit wide memory address. The segment registers are labeled CS, DS, SS, and ES.

The CS (code segment) register contains the address segment of the code (program). The DS (data segment) register contains the address segment of the area of data in memory. The SS (stack segment) register contains the address segment of the stack area. The ES (extra segment) register is an additional segment available as a second CS, DS, or SS register.

**STATUS
AND
CONTROL
REGISTERS**

There are two status and control registers which maintain the status and control of the program execution. The status and control registers are labeled IP and F.

The IP (instruction pointer) register contains the offset address, for the CS register, of the next instruction to be executed.

Flags

The 80286 microprocessor has a status and control register which contains 6 status flags and 5 control flags. Some of the assembly language instructions use these flags to conditionally change the execution path of a program. The Flags (F) register has the following format:

Bit	15	14	13	12	11	10	9	8
Flag		NT	IOPL			OF	DF	IF

Bit	7	6	5	4	3	2	1	0
Flag	TF	SF	ZF		AF		PF	CF

STATUS FLAGS

The status flags reflect conditions that result from previous instructions. Arithmetic instructions use the OF, SF, ZF, AF, PF, and CF registers.

AF

Auxiliary Carry Flag

This flag is set (i.e., equal to 1) under two conditions:

- During addition there is a carry of the low nibble to the high nibble.
- During subtraction there is a borrow from the high nibble to the low nibble.

CF	Carry Flag This flag is set when there has been a carry or a borrow to the high-order bit of the (8- or 16-bit) result.
OF	Overflow Flag When this flag is set, an arithmetic overflow has occurred and a significant digit has been lost.
PF	Parity Flag If this flag is set, the result of the operation has an even number of ones in it. Use this flag to check for data transmission errors.
SF	Sign Flag This flag is set when the high-order bit of the result is a logical 1. Since negative binary numbers are represented using two's complement notation, SF reflects the sign of the result: 0 indicates a positive number and 1 indicates a negative number.
ZF	Zero Flag This flag is set when the result of an operation is zero.

**CONTROL
FLAGS**

The control flags set processor operations for string instructions, maskable interrupts, and debugging.

DF

Direction Flag

This flag is set and cleared by the STD (Set Direction Flag) and CLD (Clear Direction Flag) instructions. If it has the value 1, SI and DI are decremented. If it has the value of 0, SI and DI are incremented. This flag is used for the following instructions: MOVS, MOVSB, MOVSW, CMPS, CMPSB, AND CMPSW.

IF

Interrupt-enable Flag

If this flag is set, external (maskable) interrupts are recognized by the microprocessor.

TF

Trap Flag

When set, the trap flag puts the system into single-step mode for the purposes of debugging. An internal interrupt is generated after each instruction so that you can inspect your program one instruction at a time.

NT

Nested Task Flag

When this flag is set, an IRET instruction is executed.

IOPL

I/O Privilege Level

This is a special field not covered within this document. It is used with the real mode of the 80286.

Addressing Modes

The 80286 microprocessor has two register and immediate addressing modes plus six memory addressing modes.

The 80286 microprocessor has these register and immediate addressing modes:

- Register Operand Addressing
- Immediate Operand Addressing.

And these memory addressing modes:

- Direct Addressing
- Register Indirect Addressing
- Based Addressing
- Indexed Addressing
- Based Indexed Addressing
- Based Indexed with Displacement Addressing.

REGISTER AND IMMEDIATE ADDRESSING

Two modes are provided for addressing when the operand is located in a register or the instruction.

Register Operand Addressing

The register operand addressing mode uses the contents of one of the registers as the operand for the instruction. The instruction can specify that either 8 bits or 16 bits are to be moved. For example:

```
MOV AX,BX ;moves 16 bits from BX to AX
           and
MOV AL,BL ;moves 8 bits from BL to AL
```

Immediate Operand Addressing

With immediate operand addressing, the operand appears in the instruction. For example,

```
MOV AX,333 ;Moves the constant
            ; value 333 into the AX
            ; register.
```

MEMORY ADDRESSING

Six modes are provided to specify the location of an operand in a memory segment. A memory operand address consists of two 16-bit components: segment and offset. The segment is supplied by a segment register either implicitly chosen by the addressing mode or explicitly chosen by a segment override prefix. The offset is calculated by summing any combination of the following three address elements:

- **Displacement** — an 8 or 16-bit immediate value contained in the instruction

- **Base** — contents of either the BX or BP base registers
- **Index** — contents of either the SI or DI index registers.

Any carry out from the 16-bit addition is ignored. Eight-bit displacements are sign extended to 16-bit values. Combinations of these three address elements define the six memory addressing modes that follow.

Direct Addressing

The direct addressing mode specifies a location in memory whose contents is used as the operand for the instruction. Example:

MOV CX,COUNT

This instruction uses the value found in the memory location designated by the symbol COUNT. Unless otherwise specified, COUNT is expected to be somewhere in the Data Segment. To specify that the operand is located in a segment other than the data segment, use the "segment override prefix:"

MOV CX,ES:COUNT

This syntax specifies that COUNT is located in the Extra Segment.

**Register
Indirect
Addressing**

With the register addressing mode, the 16-bit offset address is contained in a base or index register. That is, the offset address resides in the BX, BP, SI, or DI register. Example:

MOV AX, [SI]

The 16-bit value contained in the SI register is combined with the segment register to compute the 20-bit address of the operand to move into register AX. The register that is used to compute the address depends on which instruction you are using.

**Based
Addressing**

The operand's offset is the sum of an 8- or 16-bit displacement and the contents of a base register (BX or BP). For example:

MOV [BP+DI], AX

moves the contents of the AX register to the location specified by the sum of the BP and DI registers.

**Indexed
Addressing**

The operand's offset is the sum of an 8- or 16-bit displacement and the contents of an index register (SI or DI). For example:

MOV AX, [BX] [SI]

moves the contents of the address specified by the sum of the BX and SI registers into the AX register.

**Based
Indexed
Addressing**

The operand's offset is the sum of the contents of a base register and an index register.

**Based
Indexed with
Displacement
Addressing**

The operand's offset is the sum of a base register's contents, an index register's contents, and an 8- or 16-bit displacement.

**80286
Microprocessor
Instruction
Set**

Refer to the Hardware Reference Manual for the 80286 microprocessor instruction set.



Overview	5-3
Memory Configuration	5-4
Memory Map	5-5
Low Memory Map	5-6
ROM BIOS Data Area	5-7
ASCIIIZ Strings	5-8
Format.....	5-8
Handles	5-9
Pre-Defined Handles	5-9
File Control Blocks	5-10
Standard FCB Format	5-10
Extended FCB Format	5-14
Disk Layout	5-16
Clusters	5-16
Disk Directory	5-17
Format.....	5-17
Disk Boot Sector	5-24
Format.....	5-24

File Allocation Table (FAT)	5-27
Format.....	5-27
FAT Entries	5-28
How To Use The FAT	5-30
Example.....	5-32
Disk Formats	5-33
Formats	5-33

Overview

The purpose of this chapter is to help you locate items in memory or on disk for programming and debugging.

The first part of the chapter contains detailed memory maps of the RAM and ROM memory areas. The sections on control blocks deal with program file formats and I/O data structures. The last part of this chapter describes how data is organized on the disk.

Memory Configuration

The Maximum addressable physical memory within the PC 6300 PLUS is 16M bytes (16,384K bytes or 16,777,216 bytes). This consists of RAM and ROM installed on the motherboard and added memory on optional expansion cards.

When the PC 6300 PLUS is running (real mode) for use with MS-DOS 3.10 or 8086 programs, then only the first one megabyte is addressable.

Memory Map

The following table shows the memory address assignments of the system. The total memory address space is 16M bytes.

Address	Description
000000 09FFFF	640K bytes RAM — On motherboard
0A0000 0B7FFF	96K bytes Reserved — Optional Display Enhancement board or other video memory
0B8000 0BFFFF	32K bytes — Display Controller Board video memory
0C0000 0EFFFF	192K bytes — Reserved for ROM expansion and control
0F0000 0FFFFF	64K bytes — ROM
100000 F9FFFF	Reserved for memory expansion boards — RAM (14M bytes + 640K bytes) — accessible in protected mode only.
FA0000 FFFFFF	384K bytes of RAM on motherboard (accessible in protected mode only)

Low Memory Map

Hexadecimal addresses are in "segment:offset" format:

Address	Description
0:0000	Interrupt vectors — 0-7 8259 interrupt controller vectors — 8-FH
0:0040	BIOS interrupt vectors — 10-1FH
0:0080	MS-DOS interrupt vectors — 20-3FH
0:0100	Assignable interrupt vectors — 40-FFH
0:0400	ROM BIOS data area (a.k.a. BIOS communications area) — see next map.
0:0500	MS-DOS data area (a.k.a MS-DOS communications area)

ROM BIOS Data Area

Hexadecimal addresses are in “segment:offset” format:

Address	Description
0:0400	Hardware environment parameters (Printer and RS232C device addresses, memory size, etc.)
0:0417	Keyboard buffer and status bytes
0:043E	Floppy and fixed disk status bytes
0:0449	Video display area (current mode, color palette, cursor position, active page numbers, etc.)
0:0467	Data area for optional ROM and timer chip
0:0471	Fixed disk, I/O timeouts, and more keyboard status information
0:0488	RESERVED
0:0500	Inter-applications communications area

ASCIIZ Strings

MS-DOS versions 2.11/3.10 and later provide a set of new function calls for file I/O that are easier to use than the “traditional” calls that were used in past versions. These new calls do not utilize file control blocks. To open a diskette file, you simply provide information to identify the file in the form of an ASCIIZ string. MS-DOS returns a numeric value, a “handle”, that you use to refer to the file once you have opened it.

The older function calls that require the use of file control blocks and do not utilize ASCIIZ strings and handles are supported in MS-DOS 2.11/3.10 and later to provide upward compatibility. Use the newer function calls whenever possible. See Chapter 7 for details of function calls.

FORMAT An ASCIIZ string, also known as a pathname string, has the following format:

[d:]path\[name.ext]0

For example, to indicate file “myprog” on drive A, type:

A:\sys\mydir\myprog0
(must be followed by a byte of zeroes)

The back slash (\) is the valid path-separator character.

Handles

Several of the new function calls that support files or devices use an identifier known as a “handle” (also known as a “token”). When you create or open a file or device with these function calls, a 16-bit binary value is placed in register AX. Use this handle to refer to the file after it has been opened.

PRE- DEFINED HANDLES

The following handles are pre-defined by MS-DOS for your use. You need not open them before using them:

Handle	Description
0	Standard input device
1	Standard output device
2	Standard error output device
3	Standard auxiliary device
4	Standard printer device

Note: See your *MS-DOS By Microsoft User's Guide* for information on redirecting I/O for the first two handles.

File Control Blocks

The standard File Control Block (FCB) contains 36 bytes of file control information. The extended FCB contains 43 bytes. The extended FCB adds a 7-byte prefix to the standard FCB. The extended FCB is used to create or search for files in the disk directory that have special attributes.

Any of the MS-DOS functions which employ FCBs may use either an FCB or an extended FCB. (See Chapter 7 for a description of each MS-DOS function call.)

STANDARD A standard FCB contains the following bytes in
FCB this format:
FORMAT

Offset	Contents
0	Drive Number
1	Name
9	Filename Extension
12	Current Block Number
14	Logical Record Size
16	File Size
20	Last Change Date
22	Reserved
32	Current Record Number
33	Relative Record Number

**Drive
Number**

This byte contains the drive number. The byte contains "0" when the drive is closed. When you open the drive, the drive number is as follows:

- 0 for closed
- 1 for Drive A
- 2 for Drive B
- 3 for Drive C

Name

These eight bytes contain the name of the file or a reserved device name. For names shorter than eight bytes, the contents will be left-justified and padded with blanks. If the contents is a reserved device name (e.g., LPT1), do not include the optional colon.

**Filename
Extension**

These three bytes contain the filename extension. For extensions shorter than three bytes, the contents will be left-justified and padded with blanks. When the file has no extension, the bytes contain blanks.

**Current
Block
Number**

These two bytes indicate the current block number relative to the beginning of the file (starting at zero) and are used for sequential reads and writes.

Logical Record Size These two bytes indicate the logical record size in bytes. These bytes should be set to 80H during opening, if not set to correct value.

File Size These four bytes indicate the logical record size in bytes. The first two bytes contain the lower half followed by the upper half in the last two bytes.

Last Change Date This word (two bytes) indicates the date the file was created or last updated, whichever is latest. The first byte contains the lower half of the word followed by the upper half in the next byte. The date appears in the following format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data	YEAR								MONTH				DAY			

YEAR

To determine the year, add the data to the constant 1980. The data reflects the offset from year 1980. For example, if data is 28 (1CH) then the year is 2008.

MONTH

Determine the month by the following:

Data	Month
1	January
2	February
3	March
4	April
5	May
6	June
7	July
8	August
9	September
A	October
B	November
C	December

DAY

The data indicates the numerical day of the month (1 - 31).

EXAMPLE

The date September 18, 2047 (09/18/47) would appear as:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Data	1	0	0	0	0	1	1	1	0	0	1	1	0	0	1	0	
Entry	87H									32H							

Reserved	These bytes are reserved for use by MS-DOS.
Current Record Number	This byte indicates the current record number within the current block (0 - 127). This byte is not initialized by the open function call. Set this byte before doing sequential read or write operations.
Relative Record Number	This byte indicates the current record number relative to the beginning of the file (starting with zero).
EXTENDED FCB FORMAT	An extended FCB contains the following bytes in this format:

Offset	Contents
0	Extended Indicator (FF)
1	Reserved
6	File Attribute
7	Drive Number
8	Name
16	Filename Extension
19	Current Block Number
21	Logical Record Size
23	File Size
27	Last Change Date
29	Reserved
39	Current Record Number
40	Relative Record Number

Extended Indicator This byte contains "FF" which indicates that this is an extended FCB.

Reserved These bytes are reserved for use by MS-DOS.

File Attribute This byte indicates the file attribute. See "DISK DIRECTORY" for details.

Standard FCB The rest of the bytes in the extended FCB correspond to the standard FCB discussed earlier. See "STANDARD FCB FORMAT" for detailed information.

Disk Layout

The MS-DOS area of the disk is formatted as follows:

Reserved Area—variable size
First copy of file allocation table—variable size
Second copy of file allocation table—variable size
Root directory—variable size
File data area

CLUSTERS

Space for a file in the data area is not preallocated. The space is allocated one “cluster” at a time. A cluster consists of one or more consecutive sectors; all of the clusters for a file are “chained” together in the File Allocation Table (FAT). On diskettes formatted by MS-DOS 2.11/3.10 and later, there are two copies of the FAT kept for consistency. Should the disk develop a bad sector in the middle for the first FAT, the second is used as a backup.

Disk Directory

The **FORMAT** command builds the root directory for all disks. Its location on disk and the maximum number of entries are dependent on the media.

Since directories other than the root directory are regarded as files by MS-DOS, there is no limit to the number of files they may contain.

FORMAT A root directory contains 32 bytes in this format:

Offset	Contents
0	Name or Status
8	Filename Extension
11	File Attribute
12	Reserved
22	Last Change Time
24	Last Change Date
26	Cluster Number
28	File Size

Name or Status These eight bytes contain either a file name or status which is determined by the contents of the first byte. For names shorter than eight bytes, the contents will start in byte 1, then padded with blanks. The following codes in the first byte indicate file status information:

00H

The directory entry has never been used. This is used to limit the length of directory searches, for performance reasons.

2EH

The entry is for a directory. If the second byte is also 2EH, then the cluster field contains the cluster number of this directory's parent directory (000 if the parent directory is the root directory). Otherwise, bytes 2 through 11 are all spaces and the cluster number byte contains the cluster number of this directory.

E5H

The file was used, but it has been erased.

Any other data in the first byte represent the first character of the file name or volume name. Check the file attribute (byte 12) to determine whether bit 3 is "0" (filename) or "1" (volume name).

**Filename
Extension**

These three bytes contain the filename extension. For extensions shorter than three bytes, the contents will start in byte 9 and then be padded with blanks. When the file has no extension, the bytes contain blanks.

**File
Attribute**

This byte indicates the file attribute in this format.

Bit	7	6	5	4	3	2	1	0
Cnts	N/A	A	S	V	S	H	R	
		R	U	O	Y	I	E	
		C	B	L	S	D	A	

The system files (IBMBIO.COM and IBMDOS.COM) are marked as system files that are read-only and hidden (i.e., 0000 0111 or 07). Files can be marked hidden when they are created. Also, the read-only, hidden, system, and archive attributes may be changed through the "Change Attributes" (43H) system call.

REA - READ-ONLY FILE (BIT 0)

File is marked read-only. Any attempt to open the file for writing using the "Open File" (3DH) system call results in an error code being returned. This value can be used along with other values below. Attempts to delete the file with the "Delete File" (13H) or "Delete a Directory Entry" (41H) system call will also fail.

HID - HIDDEN FILE (BIT 1)

The file is excluded (hidden) from normal directory searches.

SYS - SYSTEM FILE (BIT 2)

The file is a system file that is excluded from normal directory searches.

VOL - VOLUME NAME (BIT 3)

The entry indicates this is a volume name, not a file. When bit 3 is set "1", the previous 11 bytes contain the volume label instead of a name with extension. The entry contains no other usable information (except date and time of creation), and may exist only in the root directory.

SUB - SUB-DIRECTORY (BIT 4)

The entry defines a sub-directory and is excluded from normal directory searches.

ARC - ARCHIVE BIT (BIT 5)

This archive bit is set "1" whenever the file has been written to and closed. It is used by BACKUP and RESTORE commands for determining whether or not a file has changed since its last backup.

EXAMPLE

Assume a file is attributed as read-only and hidden. The file attribute would be:

Bit	7	6	5	4	3	2	1	0
Data	0	0	0	0	0	0	1	1
Entry	0H					3H		

Reserved These bytes are reserved.

Last Change Time This word (two bytes) indicates the time the file was created or last updated, whichever is latest. The first byte contains the lower half of the word followed by the upper half in the next byte. The time appears in the following format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data	HOUR					MINUTE					SECOND					

HOURL

The data indicates the numerical hour of the day, based on a 24-hour clock (0 - 23).

MINUTE

The data indicates the minute of the hour (0 - 59).

SECOND

The data indicates the number of 2-second increments in the hour (0 - 30).

EXAMPLE

The time 22:38:24 would appear as:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data	1	0	1	1	0	1	0	0	1	1	0	0	1	1	0	0

Entry

B8H

|

CCH

**Last Change
Date**

This word (two bytes) indicates the date the file was created or last updated, whichever is latest. The first byte contains the lower half of the word followed by the upper half in the next byte. The date appears in the following format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data	YEAR								MONTH				DAY			

YEAR

To determine the year, add the data to the constant 1980. The data reflects the offset from year 1980. For example, if data is 28 (0011100) then the year is 2008.

MONTH

Determine the month by the following:

Data	Month
1	January
2	February
3	March
4	April
5	May
6	June
7	July
8	August
9	September
A	October
B	November
C	December

DAY

The data indicates the numerical day of the month (1 - 31).

EXAMPLE

The date September 18, 2047 (09/18/47) would appear as:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Data	1	0	0	0	0	1	1	1	0	0	1	1	0	0	1	0	
Entry	87H									32H							

Cluster Number

This word (two bytes) indicates the number of the first cluster in the file. The first cluster available for data is number 002.

Note: See "HOW TO USE THE FAT" under "FILE ALLOCATION TABLE (FAT)" for details on converting cluster numbers to logical sector numbers.

File Size

These four bytes indicate the logical record size in bytes. The first two bytes contain the lower half followed by the upper half in the last two bytes.

Disk Boot Sector

To allow different manufacturer's systems to read each other's disks, the information relating to the BPB for the disk is kept in the boot sector of the disk.

FORMAT

The format of the boot sector is:

Offset	Contents
0	JUMP to Boot Code
3	OEM Name and Version
11	Bytes per Sector
13	Sectors per Allocation Unit
14	Reserved Sectors
16	Number of FATs
17	Number of Root Directory Entries
19	Number of Sectors in Logical Image
21	Media Descriptor
22	Number of FAT Sectors
24	Sectors per Track
26	Number of Heads
28	Number of Hidden Sectors

Sectors per track, number of heads, and number of hidden sectors is optional information.

**JUMP to
Boot Code**

These three bytes contain the starting address of the Boot Code on the disk.

**OEM Name
and Version**

These eight bytes contain the Name and Version of this configuration.

**Bytes per
Sector**

These two bytes indicate the number of bytes per sector on the disk.

**Sectors per
Allocation
Unit**

This byte indicates the number of sectors per allocation unit on the disk.

**Reserved
Sectors**

These two bytes contain the number of reserved sectors on the disk.

**Number of
FATs**

This byte indicates the number of file allocation tables (FATs).

**Number of
Root
Directory
Entries**

These two bytes indicate the number of entries in the Root Directory on the disk.

**Number of
Sectors in
Logical
Image**

These two bytes indicate the number of sectors in the logical image on the disk.

Media Descriptor	This byte contains the Media Descriptor of this configuration.
Number of FAT Sectors	These two bytes indicate the number of FAT Sectors in this configuration.
Sectors per Track	These two bytes are optional and indicate the number of sectors per track on the disk. The number of sectors per track can be calculated from the total capacity of the disk.
Number of Heads	These two bytes are optional and indicate the number of heads on the drive. This is for support of different multi-head drives with the same storage capacity, but a different number of surfaces.
Number of Hidden Sectors	These two bytes are optional and indicate the number of hidden sectors on the disk. This is for support of drive-partitioning schemes.

File Allocation Table (FAT)

The following information is included primarily for system programmers who are writing installable device drivers. This section explains how MS-DOS uses the File Allocation Table to convert the clusters of a file to logical sector numbers. The driver program is then responsible for locating the logical sector on disk.

When you are writing a system utility, use the MS-DOS file management function calls for accessing files; programs that access the FAT directly are not guaranteed to be upwardly-compatible with future releases of MS-DOS.

FORMAT A FAT header contains three bytes in this format (the second and third bytes always contain "FFH"):

Offset	Contents
0	Media Descriptor
1	FFH
2	FFH

Media Descriptor This byte is the media descriptor which indicates the size and format of the disk.

The Media Descriptors for the following disk drives are:

Type	Media Descriptor	Capacity
5 ¼ in.	FE	160K
	FC	180K
	FF	320K
	FD	360K
	F9	1.2M
fixed	F8	X

Although these media bytes map directly to the first FAT bytes which must be F8-FFH, media bytes can, in general, be any value in the range 00-FFH. See "Code 2" under **COMMAND CODE** and **REQUEST HEADER** in Chapter 9 for details.

FAT ENTRIES

The FAT is an array of 12-bit entries (1.5 bytes) identifying each cluster the file has on the disk.

- The third FAT entry, which starts at byte 4, begins the mapping of the data area (cluster 002 is the first available cluster for data on a disk).
- Files are not always written sequentially on the disk. The data area is allocated one cluster at a time, skipping over clusters already allocated.

-
- The first free cluster found will be the next cluster allocated, regardless of its physical location on the disk. This permits the most efficient utilization of disk space because clusters made available by erasing files can be allocated for new files.

The following status codes may also appear in a FAT entry:

000H	Indicates the cluster is not used and is available.
FF7H	Indicates the cluster has a bad sector in it. MS-DOS will not allocate such a cluster. CHKDSK counts the number of bad bytes for its report. These bad clusters are not part of any allocation chain.
FF8 - FFFH	Indicates the last cluster of a file.
NNN	Indicates the cluster number of the next cluster in the file. The cluster number of the first cluster in the file is kept in the file's directory entry.

The FAT always begins on the first sector after the reserved sectors. If the FAT is larger than one sector, the sectors are contiguous. Two copies of the FAT are usually written for data integrity. The FAT is read into one of the MS-DOS buffers as needed (open, read, write, etc.). For performance reasons, this buffer is given a high priority so that it stays in memory as long as possible.

HOW TO USE THE FAT

Use the directory entry to find the starting cluster of the file. Next, to locate each subsequent cluster of the file:

- 1 Multiply the cluster number just used by 1.5 (each FAT entry is 1.5 bytes long).
- 2 The whole part of the product is an offset into the FAT, pointing to the entry that maps the cluster just used. That entry contains the cluster number of the next cluster of the file.
- 3 Use a MOV instruction to move the word at the calculated FAT offset into a register.
- 4 If the last cluster used was an even number, keep the low-order 12 bits of the register by ANDing it with FFFH; otherwise, keep the high-order 12 bits by shifting the register right 4 bits with a SHR instruction.

- 5 If the resultant 12 bits are FF8-FFFH, the file contains no more clusters. Otherwise, the 12 bits contain the cluster number of the next cluster in the file.

To convert the cluster to a logical sector number (relative sector, such as that used by Interrupts 25H and 26H and by DEBUG):

- 1 Subtract 2 from the cluster number.
- 2 Multiply the result by the number of sectors per cluster.
- 3 Add to this result the logical sector number of the beginning of the data area.

EXAMPLE Let's assume you have a 5¼ in. drive with a capacity of 180K bytes (media descriptor is "FC"). Further assume, the file consists of 5 clusters: 002, 003, 00B, 013, and 01C.

The FAT would be (shown as 1.5 bytes [12-bits wide] with the header [entries 1 & 2] broken into bytes for clarity):

Offset	Contents	
0	FF	Media Descriptor = FC
2	FF	FF
3	002	
4	003	
5	00B	
6	013	
7	01C	
8	FFF	

Disk Formats

On an MS-DOS disk, the clusters are arranged on diskettes to minimize head movement for multi-sided media. All of the space on a track (or cylinder) is allocated before moving on to the next track. This is accomplished by using the sequential sectors on the lowest-numbered head, then all the sectors on the next head, and so on until all sectors on all heads of the track are used. The next sector to be used will be sector 1 on head 0 of the next track.

The first byte of the FAT can sometimes be used to determine the format of the disk.

FORMATS

The following formats have been defined for the PC 6300 PLUS based on values of the first byte of the FAT.

No. sides	1	1	2	2
Tracks/side	40	40	40	40
Bytes/sector	512	512	512	512
Sectors/track	8	9	8	9
Sectors/cluster	1	1	2	2
Reserved sectors	1	1	1	1
No. FATs	2	2	2	2
Root directory entries	64	64	112	112
No. sectors	320	360	640	720
Media Descriptor	FE	FC	FF	FD
Sectors for 1 FAT	1	2	1	2



Overview	6-2
Pros And Cons For Selecting a Program File Format	6-3
Pros for .EXE	6-3
Cons for .EXE	6-3
Pros for .COM	6-4
Cons for .COM	6-4
EXE2BIN Program	6-6
Syntax	6-6
Comments	6-6
File Header Format	6-9
Format.....	6-9
Program Segment Prefix.....	6-12
Format.....	6-12
INT 20H.....	6-13
Program Loading Process.....	6-15
PSP Conditions Upon Program Initiation	6-15
Initial Conditions	6-17
Other Uses of the PSP	6-18
Relocation Process For .EXE Files	6-20

Overview

This chapter describes the MS-DOS program file formats and procedures for loading them into memory.

MS-DOS supports three program file formats: one type of .EXE file and two formats with a .COM extension.

- 1 The .EXE format is the most flexible type. .EXE files are limited in size only by the amount of user memory installed in your system. .EXE files can be loaded anywhere in memory and can cross segment boundaries.
 - Programs linked by LINK are output in .EXE format. .EXE files can be executed either by COMMAND.COM or by an EXEC function call in your program.
 - There are two types of .COM files: relocatable and non-relocatable. Both types are limited to 64K bytes in length and must be loaded on a segment boundary.
- 2 The relocatable .COM file can be invoked by COMMAND.COM just as MS-DOS commands are. This kind of a COM module takes up less diskette storage space and loads into memory more quickly than an .EXE module.
- 3 The non-relocatable .COM file is invoked by issuing the EXEC function call from your program. Since you can control whether or not this .COM module loads into memory, this is a useful technique for codes such as "Help" messages that you only want to call in on an as-needed basis. You can also use this type of .COM file to force a customized device driver to load into memory as a part of MS-DOS when you boot the system.

Pros And Cons For Selecting a Program File Format

This section is concerned with the pros and cons of selecting between a .EXE program format and the relocatable .COM program type. The non-relocatable .COM format is used as an overlay or as an installable device driver, as mentioned previously.

PROS for .EXE

The “pros” for selecting a .EXE program are:

- Can be larger than 64K bytes
- Can be loaded anywhere in memory (not restricted to starting on a segment boundary)
- Can cross segment boundaries
- Can run .EXE immediately after linking, i.e., you need not take the extra step of running EXE2BIN
- Can declare a stack segment in the assembly program.

CONS for .EXE

The “cons” for selecting a .EXE program are:

- Disk file has large “header” containing relocation information. .EXEC therefore takes more space on disk and takes longer to load into memory at execution time.

- Register set-up for using “Terminate But Stay Resident” or “Keep Process” system calls is much trickier with .EXE files than .COM.

**PROS for
.COM**

The “pros” for selecting a .COM program are:

- .COM files are smaller and faster loading because .COM does not have a file header containing relocation information.
- “Terminate But Stay Resident” or “Keep Process” system calls are relatively easy to use.

**CONS for
.COM**

The “cons” for selecting a .COM program are:

- .COM files can be no larger than one 64K bytes segment.
- .COM must be loaded on a segment boundary.
- .COM is segment-relocatable; the segment can be relocated at run time. However, all of the addresses in the program must be relative to the same segment address.

- You must use segment register manipulation with care. Because .COM files do not contain relocation information, segment addresses are not dynamically resolved at load time. For example:

Objective: To set the value in DS equal to CS.

Given: The variable CODE contains the value of CS.

The statements:

```
MOV AX, CODE
MOV DS, AX
```

do not work, because CODE will have the value that CS had at assembly time, not at run time. Use the stack to get the correct segment register addresses:

```
PUSH CS
POP DS
```

- To load another program from a .COM module, you must first free some of the memory that is allocated to your program. The code and data segments are loaded in at the lower addresses and the stack segment pointer points to the highest address. The stack “grows” toward the program. If you try to load a program into memory from a .COM module without freeing some of your initial allocation, your stack has a good chance of being overwritten.

EXE2BIN Program

EXE2BIN is an executable program available on your MS-DOS system diskette. It converts programs that are in .EXE format (as they are after having been linked) into the .COM format.

EXE2BIN generates both types of .COM files: relocatable and non-relocatable.

SYNTAX **EXE2BIN**<source><destination>

source The name of the input file.

destination The name of the output file.

COMMENTS

- In specifying the input file, everything except the filename is optional. If you do not specify a path, the default path is used. If you do not specify an extension, the default is .EXE. The input file is converted to .COM file format (memory image of the program) and placed in the output file.
- You are not required to enter any part of the output file specification. If you do not specify a drive, the drive of the input file will be used. If you do not specify an output path or filename, the input path or filename will be used. If you do not specify a filename extension in the output filename, the new file will be given an extension of .BIN.
- The input file must be in a valid .EXE format produced by the linker. The resident, or actual code and data **part** of the file must be less than 64K bytes. There must be no STACK segment.

Conversions

Two kinds of conversions are possible, depending on whether the initial CS:IP (Code Segment: Instruction Pointer) is specified in the .EXE file:

- 1 If CS:IP is specified as 0000:100H, it is assumed that the file is to be run as a .COM file with the location pointer set at 100H by the assembler statement ORG; the first 100H bytes of the file are deleted. No segment fixups are allowed, as .COM files must be segment relocatable.

Once the conversion is complete, rename the resulting file with a .COM extension. The command processor can load and execute the program in the same way as the .COM programs supplied on your MS-DOS diskettes.

- 2 If CS:IP is not specified in the .EXE file, a pure binary conversion is assumed. If segment fixups are necessary (i.e., the program contains instructions requiring segment relocation), you are prompted for the fixup value. This value is the absolute segment at which the program is to be loaded.

The resulting program is usable only when loaded at the absolute memory address specified by your application. The command processor is not capable of properly loading the program.

**Error
Messages**

File cannot be converted

CS:IP does not meet either of the criteria specified above, or it meets the .COM file criterion but has segment fixups. This message is also displayed if the file is not a valid executable file.

File not found

The file is not on the diskette specified.

Insufficient memory

There is not enough memory to run EXE2BIN.

File creation error

EXE2BIN cannot create the output file. Run CHKDSK to determine if the directory is full or if some other condition caused the error.

Insufficient disk space

There is not enough disk space to create a new file.

**Fixups needed - base segment
(hexadecimal):**

The source (.EXE) file contained information indicating that a load segment is required for the file. Specify the absolute segment address at which the finished module is to be located.

File cannot be converted

The input file is not in the correct format.

WARNING-Read error in EXE file

Amount read less than size in header. This is a warning message only.

File Header Format

The .EXE files produced by LINK contain two parts:

- Control and relocation information
- The load module.

The control and relocation information is at the beginning of the file in an area called the header. The load module immediately follows the header.

Note: The .COM files do not have file headers.

FORMAT The File Header contains 14 words in this format (offset is in bytes):

Offset	Contents
0	4D5AH
2	# of Bytes
3	Size of File
6	# of Relocation Entries
8	Size of Header
AH	Minimum #
CH	Maximum #
EH	Stack Value
10H	SP Value
12H	Sum of Words
14H	IP Value
16H	CS Value
18H	Relative Offset
D	# of Overlays

# of Bytes	This word indicates the number of bytes contained in the last page and is used for reading overlays.
Size of File	This word indicates the size of the file in 512-byte pages, including the header.
# of Relocation Entries	This word indicates the number of relocation entries in the table.
Size of Header	This word indicates the size of the header in 16-byte blocks. This is used to locate the beginning of the load module in the file.
Minimum#	This word indicates the minimum number of 16-byte blocks required above the end of the loaded program.
Maximum#	This word indicates the maximum number of 16-byte blocks required above the end of the loaded program.
<p>Note: If both the Minimum# and Maximum# parameters are 0, then the program will be loaded as high as possible.</p>	
Stack Value	This word contains the initial value to be loaded into the Stack segment (SS register) before starting program execution. This must be adjusted by relocation.

SP Value	This word contains the initial value to be loaded into the Stack Pointer (SP) before starting program execution.
Sum of Words	This word indicates the negative sum of all the words in the file.
IP Value	This word contains the initial value to be loaded into the Index Pointer (IP) before starting program execution.
CS Value	This word contains the initial value to be loaded into the Code segment (CS register) before starting program execution. This must be adjusted by relocation.
Relative Offset	This word contains the relative byte offset from the beginning of the run file to the relocation table.
# of Overlays	This word indicates the number of overlays generated by the LINK program.

Program Segment Prefix

Unless you specify otherwise when linking your program, MS-DOS loads your program in the lowest memory address available, immediately following the MS-DOS code.

This occurs whether the program loads as a result of your entering its name at the MS-DOS prompt or through your use of the EXEC function call. The area into which your program is loaded is called the Program Segment.

MS-DOS requires control information for each running program: it builds a Program Segment Prefix and places it at offset 0 within the program segment. The Program Segment Prefix is 50H words long, so your program is loaded at relative address 100H.

FORMAT The Program Segment Prefix contains 50H words in this format (offset is in bytes):

Offset	Contents
0	INT 20H
1	Allocation Block
2	Reserved
3	# of Bytes
4	N/A
5	Terminate Adrs
7	Control-C Exit Adrs
9	Error Exit Adrs
BH	Reserved
16H	Reserved for MS-DOS
28H	Function Dispatch Call
2EH	Area 1
36H	Area 2
40H	Parameter Area

INT 20H	This word contains the return address used by INT 20H.
Allocation Block	This word contains the segment address of the allocatable memory following this program. <i>Note:</i> If this program calls a memory management function to get more memory, this is its starting address.
# of Bytes	This word contains the number of bytes in this program segment.
Terminate Adrs	These two words contain the terminate address for the IP and CS registers — IP terminate address is first.
Control-C Exit Adrs	These two words contain the CTRL C break exit address for the IP and CS registers — IP CTRL C break exit address is first.
Error Exit Adrs	These two words contain the error exit address for the IP and CS registers — IP error exit address is first.
Reserved for MS-DOS	This parameter contains the segment address of the environment and is reserved for use by MS-DOS only.
Function Dispatch Call	This parameter contains the code used to call the function dispatcher for MS-DOS (INT 21H) interrupts.

Area 1 This parameter is the formatted parameter area 1. It is formatted as a standard unopened FCB.

Area 2 This parameter is the formatted parameter area 2. It is formatted as a standard unopened FCB.

Note: This area is overlaid if the FCB in area 1 is opened.

Parameter Area This parameter is the unformatted parameter area, also known as the default disk transfer area. Offset 40H contains the count of argument characters that follow the command name. 41H contains the argument characters themselves.

Program Loading Process

PSP CONDITIONS UPON PROGRAM INITIATION

When a program receives control, the following conditions are in effect:

- The segment address of the passed environment is contained at offset 16H in the Program Segment Prefix. The environment is a series of ASCII strings (totaling less than 32K bytes) in the form:

NAME = parameter

Each string is terminated by a byte of zeros, and the set of strings is terminated by another byte of zeros. The environment built by the command processor contains at least a COMSPEC= string (the parameters on COMSPEC define the path used by MS-DOS to locate COMMAND.COM on disk). The last PATH and PROMPT commands issued will also be in the environment, along with any environment strings defined with the MS-DOS SET command.

The environment that is passed is a copy of the invoking process environment. If your application uses a "terminate and stay resident" concept, be aware that the copy of the environment passed to you is static. That is, it will not change even if subsequent SET, PATH or PROMPT commands are issued.

- Offset 28H in the Program Segment Prefix contains code to call the MS-DOS function dispatcher. By placing the desired function request number in AH, a program can issue a far call to offset 28H to invoke an MS-DOS function, rather than issuing an Interrupt 21H. Since this is a call and not an interrupt, MS-DOS may place any code appropriate to making a system call at this position. This makes the process of calling the system portable.
- The Disk Transfer Address (DTA) is set to 40H (default DTA in the Program Segment Prefix).
- File control blocks at 2EH and 36H are formatted from the first two parameters typed when the command was entered. If either parameter contains a pathname, then the corresponding FCB contains only the valid drive number. The filename field will not be valid.
- An unformatted parameter area at 41H contains all the characters typed after the command (including leading and embedded delimiters), with the byte at 40H set to the number of characters. If the <, >, or parameters were typed on the command line, they (and the filenames associated with them) do not appear in this area; redirection of standard input and output is transparent to applications.
- Offset 3 (one word) contains the number of bytes available in the segment.

- Register AX indicates whether or not the drive specifiers (entered with the first two parameters) are valid, as follows:

AL=FF if the first parameter contained an invalid drive specifier (otherwise AL=00)

AH=FF if the second parameter contained an invalid drive specifier (otherwise AH=00)

- Offset 1 (one word) contains the segment address of the first byte of unavailable memory. Programs must not modify addresses beyond this point unless they were obtained by allocating memory via the Allocate Memory system call (Function Request 48H).

INITIAL CONDITIONS

.EXE Programs

The initial conditions for .EXE programs are:

- DS and ES registers are set to point to the Program Segment Prefix.
- CS, IP, SS, and SP registers are set to the values passed by LINK.

.COM Programs

The initial conditions for .COM programs are:

- All four segment registers contain the segment address of the initial allocation block that starts with the Program Segment Prefix control block.

- All of user memory is allocated to the program. If the program invokes another program through Function Request 4BH, it must first free some memory through the Set Block (4AH) function call, to provide space for the program being executed.
- The Instruction Pointer (IP) is set to 100H.
- The Stack Pointer register is set to the end of the program's segment. The segment size at offset 6 is reduced by 100H to allow for a stack of that size.
- A word of zeros is placed on top of the stack. This allows your program to exit to COMMAND.COM by doing a RET instruction last. Make sure, however, to maintain your stack and code segments.

OTHER USES OF THE PSP

The PSP contains the mechanism for program termination. One of these four techniques must be used to terminate your programs:

- 1 A long jump to offset 0 in the Program Segment Prefix.
- 2 By issuing an INT 20H with CS:0 pointing at the PSP.
- 3 By issuing an INT 21H with register AH = 0 with CS:0 pointing at the PSP, or AH = 4CH and no restrictions on CS.

- 4 By a long call to location 50H in the Program Segment Prefix with AH = 0 or Function Request 4CH.

Note: It is the responsibility of all programs to ensure that the CS register contains the segment address of the Program Segment Prefix when terminating via any of these methods, except Function Request 4CH. For this reason, Function Request 4CH is the preferred method.

Relocation Process For .EXE Files

The relocation table follows the formatted area described above. This table consists of a variable number of relocation items.

Each relocation item contains two fields: a 2-byte offset value, followed by a 2-byte segment value. These two fields contain the offset into the load module of a word which requires modification before the module is given control. The following steps describe this process:

- 1 The formatted part of the header is read into memory. Its size is 1BH.
- 2 A portion of memory is allocated depending on the size of the load module and the allocation numbers (0A-0B and 0C-0D). MS-DOS attempts to allocate FFFFH paragraphs. This will always fail, returning the size of the largest free block. If this block is smaller than minalloc and loadsize, then there will be no memory error. If this block is larger than maxalloc and loadsize, MS-DOS will allocate (maxalloc + loadsize). Otherwise, MS-DOS will allocate the largest free block of memory.
- 3 A Program Segment Prefix is built in the lowest part of the allocated memory.
- 4 The load module size is calculated by subtracting the header size from the file size. Offsets 04-05 and 08-09 can be used for this calculation. The actual size is downward-adjusted based on the contents of offsets 02-03. Based on the setting of the high/low loader switch, an appropriate segment is determined at which to load the load module. This segment is called the start segment.

- 5 The load module is read into memory beginning with the start segment.
- 6 The relocation table items are read into a work area.
- 7 Each relocation table item segment value is added to the start segment value. This calculated segment, plus the relocated item offset value, points to a word in the load module to which is added the start segment value. The result is placed back into the word in the load module.
- 8 Once all relocation items have been processed, the SS and SP registers are set from the values in the header. Then, the start segment value is added to SS. The ES and DS registers are set to the segment address of the Program Segment Prefix. The start segment value is added to the header CS register value. The result, along with the header IP value, is the initial CS:IP to transfer to before starting execution of the program.



Overview	7-7
Table of Interrupts	7-8
Numerical Table of Functions	7-9
Alphabetical Table of Functions	7-12
Programming Considerations	7-15
Calling from Macro Assembler	7-15
Calling from a High-Level Language	7-15
GW BASIC	7-15
Interrupts (INT)	7-16
INT 20H - Program Terminate	7-17
INT 21H - Function Request	7-19
INT 22H - Terminate Address	7-20
INT 23H - Control-C Exit Address	7-21
INT 24H - Fatal Error Abort Address	7-22
INT 25H - Absolute Disk Read	7-27
INT 26H - Absolute Disk Write	7-30
INT 27H - Terminate but Stay Resident	7-33
Functions	7-35
Registers	7-35

00H - Terminate Program	7-36
01H - Read Keyboard and Echo.....	7-38
02H - Display Character	7-40
03H - Auxiliary Input	7-41
04H - Auxiliary Output.....	7-42
05H - Print Character	7-44
06H - Direct Console I/O	7-46
07H - Direct Console Input.....	7-49
08H - Read Keyboard	7-51
09H - Display String	7-53
0AH - Buffered Keyboard Input	7-54
0BH - Check Keyboard Status	7-57
0CH - Flush Buffer, Read Keyboard	7-59
0DH - Reset Disk.....	7-61
0EH - Select Disk.....	7-63
0FH - Open File	7-64
10H - Close File	7-67
11H - Search for First Entry	7-69
12H - Search for Next Entry	7-72
13H - Delete File	7-74

14H - Sequential Read	7-76
15H - Sequential Write	7-79
16H - Create File	7-82
17H - Rename File	7-84
19H - Get Current Disk	7-86
1AH - Set Disk Transfer Address	7-87
1BH - Get Default Drive Data	7-89
1CH - Get Drive Data	7-91
21H - Random Read	7-93
22H - Random Write	7-96
23H - Get File Size	7-99
24H - Set Relative Record	7-102
25H - Set Interrupt Vector	7-104
26H - Create New PSP	7-106
27H - Random Block Read	7-107
28H - Random Block Write	7-110
29H - Parse File Name	7-113
2AH - Get Date	7-117
2BH - Set Date	7-119
2CH - Get Time	7-121

2DH - Set Time	7-123
2EH - Set/Reset Verify Flag	7-125
2FH - Get Disk Transfer Address	7-127
30H - Get MS-DOS Version	7-128
31H - Keep Process	7-129
33H - Control-C Check	7-130
35H - Get Interrupt Vector.....	7-132
36H - Get Disk Free Space	7-134
38H - Get/Set Country Data.....	7-136
39H - Create Directory.....	7-142
3AH - Remove Directory	7-144
3BH - Change the Current Directory	7-146
3CH - Create Handle	7-148
3DH - Open Handle.....	7-150
3EH - Close Handle.....	7-152
3FH - Read Handle	7-154
40H - Write Handle.....	7-156
41H - Delete a Directory Entry	7-158
42H - Move File Pointer.....	7-160
43H - Get/Set Attributes	7-163

4400H, 4401H - IOCTL Data	7-165
4402H, 4403H - IOCTL Character	7-168
4404H, 4405H - IOCTL Block	7-170
4406H, 4407H - IOCTL Status	7-172
4408H - IOCTL is Changeable	7-174
4409H - IOCTL is Redirected Block	7-176
440AH - IOCTL is Redirected Handle	7-178
440BH - IOCTL Retry	7-180
45H - Duplicate File Handle	7-182
46H - Force Duplicate of a Handle	7-184
47H - Get Current Directory Path	7-186
48H - Allocate Memory	7-188
49H - Free Allocated Memory	7-190
4AH - Set Block	7-192
4B00H - Load and Execute a Program	7-194
4B03H - Load Overlay	7-198
4CH - End Process	7-201
4DH - Get Return Code of Child Process	7-203

4EH - Find First File	7-204
4FH - Find Next File	7-206
54H - Get Verify State	7-208
56H - Change Directory Entry	7-209
57H - Get/Set Date/Time of File	7-211
58H - Get/Set Allocation Strategy	7-214
59H - Get Extended Error	7-217
5AH - Create Temporary File	7-221
5BH - Create New File	7-224
5C00H - Lock	7-226
5C01H - Unlock	7-229
5E00H - Get Machine Name	7-232
5E02H - Printer Setup	7-234
5F02H - Get Assign List Entry	7-236
5F03H - Make Assign List Entry	7-239
5F04H - Cancel Assign List Entry	7-242
62H - Get PSP	7-244
General Macros	7-245

Overview

System Calls are procedures used to interface with I/O or to manage memory. They can be accessed from utility programs written in assembly language, and from some high-level languages. Using these system calls frees the programmer from having to perform primitive functions, and makes it easier to write machine-independent programs.

There are two types of system calls: interrupts and function requests. This chapter describes the environments from which these routines can be called, how to call them, and the processing performed by each.

Each system call description contains the PURPOSE, CALL, RETURN, MACRO, COMMENTS, and EXAMPLE. The PURPOSE indicates why you would use that particular function. CALL indicates the initial conditions that must be established before you can invoke the interrupt or function. RETURN indicates what values, if any, are present after the system call is performed. MACRO indicates how to set up the system call as a macro instruction. You may find the macros a convenient way to include system calls in your assembly language programs. COMMENTS is a further elaboration on the system call.

General macro definitions are listed at the end of the chapter.

Note: Unless otherwise stated, all numbers in the system call descriptions, both text and code, are in hexadecimal.

Table of Interrupts

INTERRUPTS (NUMERICAL ORDER)

Interrupt	Description
20H	Program Terminate
21H	Function Request
22H	Terminate Address
23H	Control-C Exit Address
24H	Fatal Error Abort Address
25H	Absolute Disk Read
26H	Absolute Disk Write
27H	Terminate but Stay Resident
28-40H	RESERVED FOR MS-DOS

INTERRUPTS (ALPHABETICAL ORDER)

Description	Interrupt
Absolute Disk Read	25H
Absolute Disk Write	26H
Control-C Exit Address	23H
Fatal Error Abort Address	24H
Function Request	21H
Program Terminate	20H
RESERVED FOR MS-DOS	28-40H
Terminate Address	22H
Terminate but Stay Resident	27H

Numerical Table of Functions

FUNCTION	NAME
00H	Terminate Program
01H	Read Keyboard and Echo
02H	Display Character
03H	Auxiliary Input
04H	Auxiliary Output
05H	Print Character
06H	Direct Console I/O
07H	Direct Console Input
08H	Read Keyboard
09H	Display String
0AH	Buffered Keyboard Input
0BH	Check Keyboard Status
0CH	Flush Buffer, Read Keyboard
0DH	Reset Disk
0EH	Select Disk
0FH	Open File
10H	Close File
11H	Search for First Entry
12H	Search for Next Entry
13H	Delete File
14H	Sequential Read
15H	Sequential Write
16H	Create File
17H	Rename File
19H	Get Current Disk
1AH	Set Disk Transfer Address
1BH	Get Default Drive Data
1CH	Get Drive Data
21H	Random Read
22H	Random Write
23H	Get File Size
24H	Set Relative Record

FUNCTION	NAME
25H	Set Interrupt Vector
26H	Create New PSP
27H	Random Block Read
28H	Random Block Write
29H	Parse File Name
2AH	Get Date
2BH	Set Date
2CH	Get Time
2DH	Set Time
2EH	Set/Reset Verify Flag
2FH	Get Disk Transfer Address
30H	Get MS-DOS Version
31H	Keep Process
33H	Control-C Check
35H	Get Interrupt Vector
36H	Get Disk Free Space
38H	Get/Set Country Data
39H	Create Directory
3AH	Remove Directory
3BH	Change the Current Directory
3CH	Create Handle
3DH	Open Handle
3EH	Close Handle
3FH	Read Handle
40H	Write Handle
41H	Delete a Directory Entry
42H	Move File Pointer
43H	Get/Set Attributes
4400H	IOCTL Data (Get)
4401H	IOCTL Data (Set)
4402H	IOCTL Character (Receive)
4403H	IOCTL Character (Send)
4404H	IOCTL Block (Send)
4405H	IOCTL Block (Receive)
4406H	IOCTL Status (Input)
4407H	IOCTL Status (Output)
4408H	IOCTL is Changeable

FUNCTION	NAME
4409H	IOCTL is Redirected Block
440AH	IOCTL is Redirected Handle
440BH	IOCTL Retry
45H	Duplicate File Handle
46H	Force Duplicate of a Handle
47H	Get Current Directory Path
48H	Allocate Memory
49H	Free Allocated Memory
4AH	Set Block
4B00H	Load and Execute a Program
4B03H	Load Overlay
4CH	End Process
4DH	Get Return Code of Child Process
4EH	Find First File
4FH	Find Next File
54H	Get Verify State
56H	Change Directory Entry
57H	Get/Set Date/Time of File
58H	Get/Set Allocation Strategy
59H	Get Extended Error
5AH	Create Temporary File
5BH	Create New File
5C00H	Lock
5C01H	Unlock
5E00H	Get Machine Name
5E02H	Printer Setup
5F02H	Get Assign List Entry
5F03H	Make Assign List Entry
5F04H	Cancel Assign List Entry
62H	Get PSP

Alphabetical Table of Functions

NAME	FUNCTION
Allocate Memory	48H
Auxiliary Input	03H
Auxiliary Output	04H
Buffered Keyboard Input	0AH
Cancel Assign List Entry	5F04H
Change Directory Entry	56H
Change the Current Directory	3BH
Check Keyboard Status	0BH
Close File	10H
Close Handle	3EH
Control-C Check	33H
Create Directory	39H
Create Handle	3CH
Create File	16H
Create New File	5BH
Create New PSP	26H
Create Temporary File	5AH
Delete a Directory Entry	41H
Delete File	13H
Direct Console Input	07H
Direct Console I/O	06H
Display Character	02H
Display String	09H
Duplicate File Handle	45H
End Process	4CH
Find First File	4EH
Find Next File	4FH
Flush Buffer, Read Keyboard	0CH
Force Duplicate of a Handle	46H
Free Allocated Memory	49H
Get/Set Allocation Strategy	58H
Get Assign List Entry	5F02H

NAME	FUNCTION
Get/Set Attribute	43H
Get/Set Country Data	38H
Get Current Directory	47H
Get Current Disk	19H
Get Date	2AH
Get/Set Date/Time of File	57H
Get Default Drive Data	1BH
Get Disk Free Space	36H
Get Disk Transfer Address	2FH
Get Drive Data	1CH
Get Extended Error	59H
Get File Size	23H
Get Interrupt Vector	35H
Get Machine Name	5E00H
Get MS-DOS Version	30H
Get Return Code of Child Process	4DH
Get PSP	62H
Get Time	2CH
Get Verify State	54H
IOCTL Block (Receive)	4405H
IOCTL Block (Send)	4404H
IOCTL Character (Send)	4403H
IOCTL Character (Receive)	4402H
IOCTL Data (Get)	4400H
IOCTL Data (Set)	4401H
IOCTL is Changeable	4408H
IOCTL is Redirected Block	4409H
IOCTL is Redirected Handle	440AH
IOCTL Status (Input)	4406H
IOCTL Status (Output)	4407H
IOCTL Retry	440BH
Keep Process	31H
Load and Execute a Program	4B00H
Load Overlay	4B03H
Lock	5C00H
Make Assign List Entry	5F03H
Move File Pointer	42H

NAME	FUNCTION
Open File	0FH
Open Handle	3DH
Parse File Name	29H
Print Character	05H
Printer Setup	5E02H
Random Block Read	27H
Random Block Write	28H
Random Read	21H
Random Write	22H
Read Handle	3FH
Read Keyboard	08H
Read Keyboard and Echo	01H
Remove Directory	3AH
Rename File	17H
Reset Disk	0DH
Search for First Entry	11H
Search for Next Entry	12H
Select Disk	0EH
Sequential Read	14H
Sequential Write	15H
Set Block	4AH
Set Date	2BH
Set Disk Transfer Address	1AH
Set Interrupt Vector	25H
Set Relative Record	24H
Set Time	2DH
Set/Reset Verify Flag	2EH
Terminate Program	00H
Unlock	5C01H
Write Handle	40H

Programming Considerations

System calls can be invoked from the Macro Assembler, from GWBASIC, or from high-level languages like PASCAL and FORTRAN.

CALLING FROM MACRO ASSEMBLER

The system calls can be invoked from the Macro Assembler simply by moving any required data into registers and issuing an interrupt. Some of the calls destroy registers, so you may have to save registers before using a system call.

CALLING FROM A HIGH- LEVEL LANGUAGE

The system calls can be invoked from any high-level language whose modules can be linked with assembly-language modules. The system calls are embedded in the assembly language module.

GWBASIC

Different techniques are used to invoke system calls from the compiler and interpreter. From the interpreter, the CALL statement or USR function can be used to execute 80286 microprocessor object code.

The **BLOAD** and **BSAVE** (GWBASIC) commands are used for loading and saving machine language programs. These are then called, using the CALL statement.

The USR function calls an indicated machine language subroutine. The starting address of the subroutine must first be specified in a DEF USR statement.

Interrupts (INT)

Interrupts 28 through 40H are reserved for MS-DOS. The rest are for user programs. The table of interrupt routine addresses (vectors) is maintained in locations 80H-FCH. User programs should only issue Interrupts 20H, 21H, 25H, 26H, and 27H. (Functions Requests 4CH and 31H are the preferred method for Interrupts 20H and 27H for versions of MS-DOS that are 2.11/3.10 and later.)

Interrupts 22H, 23H, and 24H are not interrupts that can be used by user programs; they are simply locations where a segment and offset address are stored. For a discussion, see the following sections on Address Interrupts in this chapter.

- Terminate Address (Interrupt 22H)
- `CTRL`C Exit Address (Interrupt 23H)
- Fatal Error Abort Address (Interrupt 24H)

Interrupts are issued under the specified circumstance. You can change any of these addresses with Function Request 25H (Set Vector) if you prefer to write your own interrupt handlers.

A detailed description of each interrupt follows in numerical order.

Note: The abbreviation "INT" is used to differentiate the interrupts from the functions discussed in this chapter.

INT 20H - Program Terminate

PURPOSE Use to terminate a program correctly.

CALL CS — Segment address of Program
 Segment Prefix (PSP)

RETURN None

MACRO `terminate macro`
 `int 20H`
 `endm`

COMMENTS All open file handles are closed and the disk cache is cleared. This interrupt is almost always used in old .COM file for termination.

The CS register must contain the segment address of the Program Segment Prefix before you call this interrupt.

The following exit addresses are restored from the Program Segment Prefix:

Exit Address	Offset
Program Terminate	0AH
Control-C	0EH
Critical Error	12H

All file buffers are flushed to disk.

Note: Close all files that have changed in length before issuing this interrupt. If a changed file is not closed, its length is not recorded correctly in the directory. See Functions 10H and 3EH for a description of the Close File system calls.

Interrupt 20H is provided for compatibility with versions of MS-DOS prior to 2.0. New programs should use Function Request 4CH (Terminate a Process).

EXAMPLE

This example terminates a program:

```
;CS must equal PSP value given at start
;(ES and DS values)
    int 20H
;There is no return from this interrupt
```


INT 21H - Function Request

PURPOSE	Use to initiate the function specified by the AH register.
CALL	AH — Function number (Other registers are specified in individual functions)
RETURN	As specified in individual functions
MACRO	None
COMMENTS	The AH register must contain the number of the function request. See the following section on FUNCTION REQUESTS, in this chapter, for a description of the MS-DOS system functions requests.

Note: No macro is defined for this interrupt, because all function descriptions in this chapter that define a macro include Interrupt 21H.

EXAMPLE This example calls the 2CH (Get Time) function:

```
mov    ah,2CH    ;Get Time is Function 2CH
int     21H      ;THIS INTERRUPT
end
```

INT 22H - Terminate Address

For every program, the operating system must know where the end (Terminate Address) is, so it can dispose of the program and load the next one (if the program calls another).

When a program terminates, control transfers to the address at offset 0AH of the Program Segment Prefix. This address is copied into the Program Segment Prefix, from the Interrupt 22H vector, when the segment is created.

INT 23H - Control-C Exit Address

If the user types **CTRL**C during keyboard input or display output, control transfers to the INT 23H vector in the interrupt table. This address is copied into the Program Segment Prefix, from the Interrupt 23H vector, when the segment is created.

If the **CTRL**C routine preserves all registers, it can end with an IRET instruction (return from interrupt) to continue program execution. When the interrupt occurs, all registers are set to the value they had when the original call to MS-DOS was made. There are no restrictions on what a **CTRL**C handler can do—including MS-DOS function calls—so long as the registers are unchanged if IRET is used.

If Function 09H or 0AH (Display String or Buffered Keyboard Input) is interrupted by **CTRL**C, the 3-byte sequence 03H-0DH-0AH (EXT-CR-LF) is sent to the display and the function resumes at the beginning of the next line.

If the program creates a new segment and loads a second program that changes the **CTRL**C address, termination of the second program must restore the **CTRL**C address to its value before execution of the second program.

INT 24H - Fatal Error Abort Address

If a fatal disk error occurs during execution of one of the Disk I/O function calls, control transfers to the INT 24H vector in the vector table. This address is copied into the Program Segment Prefix, from the Interrupt 24H vector, when the segment is created.

BP:SI contains the address of a Device Header Control Block from which additional information can be retrieved.

Note: Interrupt 24H is not issued if the failure occurs during execution of Interrupt 25H (Absolute Disk Read) or Interrupt 26H (Absolute Disk Write). These errors are usually handled by the MS-DOS error routine in COMMAND.COM that retries the disk operation, then gives the user the choice of aborting, retrying the operation, or ignoring the error. The following topics give you the information you need about interpreting the error codes, managing the registers and stack, and controlling the system's response to the error in order to write your own error-handling routines.

ERROR CODES

When an error-handling program gains control from Interrupt 24H, the AX and DI registers can contain codes that describe the error. If Bit 7 of AH is 1, the error is either a bad image of the File Allocation Table or an error which has occurred on a character device. The device header passed in BP:SI can be examined to determine which case exists.

If the attribute byte high order bit indicates a block device, then the error was a bad FAT. Otherwise, the error is on a character device.

The following are error codes for Interrupt 24H:

Error Code	Description
0	Attempt to write on write-protected disk
1	Unknown unit
2	Drive not ready
3	Unknown command
4	CRC data error
5	Bad drive request structure length
6	Seek error
7	Unknown media type
8	Sector not found
9	Printer out of paper
A	Write fault
B	Read fault
C	General failure.

The user stack will be in effect (the first item described below is at the top of the stack), and will contain the following from top to bottom:

Register	Description
IP	MS-DOS registers from issuing INT 24H
CS	
FLAGS	
AX	User registers at time of original INT 21H request
BX	
CX	
DX	
SI	
DI	
BP	
DS	
ES	
IP	From the original INT 21H from the user to MS-DOS.
CS	
FLAGS	

The registers are set such that if an IRET is executed, MS-DOS will respond according to AL as follows:

- AL = 0 — Ignore the error
- AL = 1 — Retry the operation
- AL = 2 — Terminate the program via
 INT 23H
- AL = 3 — Fail the system call in progress.

The following bits in AH indicate which responses (ABORT, RETRY, IGNORE, and FAIL) are allowed if the error is a block or character device error. An ABORT response is always allowed and a "1" in the following locations indicates allowed:

Bit 5 = IGNORE

Bit 4 = RETRY

Bit 3 = FAIL.

Consider the following rules when a desired response is not allowed:

- If IGNORE is NOT allowed, change response to FAIL.
- If RETRY is NOT allowed, change response to FAIL.
- If FAIL is NOT allowed (even above), change response to ABORT.

Before giving this routine control for disk errors, MS-DOS performs five retries.

- For disk errors, this is taken only for errors occurring during an Interrupt 21H. It is not used for errors during Interrupts 25H or 26H.
- This routine is entered is a disabled state.
- The SS, SP, DS, EX, BX, CX, and DX registers must be preserved.

- The interrupt handler should refrain from using MS-DOS function calls. If necessary, it may use calls 01H through 0CH. Use of any other call will destroy the MS-DOS stack and will leave MS-DOS in an unpredictable state.
- The interrupt handler must not change the contents of the device header.
- If the interrupt handler will handle errors rather than returning to MS-DOS, it should restore the application program's registers from the stack, remove all but the last three words on the stack, then issue an IRET. This will return to the program immediately after the INT 21H that experienced the error. Note that if this is done, MS-DOS will be in an unstable state until a function call higher than 0CH is issued.
- Some errors are mapped to one of the 0 through CH error codes; therefore, the true error is available via Get Extended Error.

INT 25H - Absolute Disk Read

PURPOSE	Used to transfer control to the device driver. The sectors are read from the disk to the Disk Transfer Address.
----------------	---

CALL	AL — Drive number
	DS:BX — Disk Transfer Address
	CX — Number of sectors
	DX — Beginning relative sector

RETURN	FLAGS
	CF = 0 if successful
	CF = 1 if not successful
	AL — Error code if CF = 1

```
MACRO      abs_disk_read macro disk,buffer,num_sectors,
           start
           mov     al,disk
           mov     bx,offset buffer
           mov     cx,num_sectors
           mov     dx,start
           int     25H
           endm
```

COMMENTS These registers must contain the following:

AL	Drive number (0 = A, 1 = B, etc.)
BX	Offset of Disk Transfer Address (from segment address in DS).
CX	Number of sectors to read.
DX	Beginning relative sector.

This interrupt transfers control to the device driver. The number of sectors specified in CX is read from the disk to the Disk Transfer Address. Its requirements and processing are identical to Interrupt 26H, except data is read rather than written.

Note: All registers except the segment registers are destroyed by this call. Be sure to save any register your program uses before issuing the interrupt.

The system pushes the flags at the time of the call; they are still there upon return. (This is necessary because data is passed back in the flags). Be sure to pop the stack upon return to restore your stack pointer at the point of invocation.

If the disk operation was successful, CF is 0. If the disk operation was not successful, CF is 1 and AL contains the MS-DOS error code (see Interrupt 24H earlier in this section for the codes and their meaning).

EXAMPLE

The following program copies the contents of a single-sided disk in drive A to the disk in drive B, it uses a buffer of 32K bytes:

Note: This example must have functions 08H and 09H loaded to work properly.

```

prompt      db  "Source in A, destination in B", 0DH, 0AH
            db  "Press any key to start. $"
start       dw  0
buffer      db  40H dup (200H dup (?))    ;64 sectors
            .
            .
            .
            display prompt                ;Function 09H
            read_kbd                      ;Function 08H
            mov cx,5                      ;copy 5 groups
copy:       push cx                      ;save loop counter
            abs_disk_read 0,buffer,40H,start ;THIS INTERRUPT
            abs_disk_write 1,buffer,40H,start ;INT 26H
            add start,40H                ;do next 64 sectors
            pop cx                      ;restore loop counter
            loop copy

```

INT 26H - Absolute Disk Write

PURPOSE	Use to transfer control to the device driver. The sectors are written from the Disk Transfer Address to the disk.
----------------	---

CALL	AL — Drive number
	DS:BX — Disk Transfer Address
	CX — Number of sectors
	DX — Beginning relative sector

RETURN	FLAGS
	CF = 0 if successful
	CF = 1 if not successful
	AL — Error code if CF = 1

```
MACRO      abs_disk_write macro disk,buffer,num_sectors,
           start
           mov     al,disk
           mov     bx,offset buffer
           mov     cx,num_sectors
           mov     dx,start
           int     26H
           endm
```

COMMENTS These registers must contain the following:

AL	Drive number (0 = A, 1 = B, etc.).
BX	Offset of Disk Transfer Address (from segment address in DS).
CX	Number of sectors to write.
DX	Beginning relative sector.

This interrupt transfers control to the device driver. The number of sectors specified in CX is written from the Disk Transfer Address to the disk. Its requirements and processing are identical to Interrupt 25H, except data is written to the disk rather than read from it.

Note: All registers except the segment registers are destroyed by this call. Be sure to save any registers your program uses before issuing the interrupt.

The system pushes the flags at the time of the call; they are still there upon return. (This is necessary because data is passed back in the flags). Be sure to pop the stack upon return to restore your stack pointer at the point of invocation.

If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains the MS-DOS error code (see Interrupt 24H earlier in this section for the codes and their meaning).

EXAMPLE

The following program copies the contents of a single-sided disk in drive A to the disk in drive B, verifying each write (uses a buffer of 32K bytes):

Note: This example must have functions 08H, 09H, and 2EH loaded to work properly.

```
off          equ 0
on           equ 1
.
.
.
prompt       db "Source in A, target in B", 0DH, 0AH
              db "Any key to start. $"
start        dw 0
buffer       db 40H dup (200H dup (??)) ;64 sectors
.
.
.
int_26H:     display                      ;Function 09H
              read_kbd                    ;Function 08H
              verify on                   ;Function 2EH
              mov cx,5                     ;copy 5 groups
copy:        push cx                      ;save loop counter
              abs_disk_read 0,buffer,40H,start ;INT 25H
              abs_disk_write 1,buffer,64,start ;THIS INTERRUPT
              add start,40H                ;do next 64 sectors
              pop cx                       ;restore loop counter
              loop copy
              verify off                   ;Function 2EH
              end
```

INT 27H - Terminate but Stay Resident

PURPOSE Use to terminate a program and allow the code to remain resident.

CALL CS:DX — First byte following last byte of code

RETURN None

MACRO

```
stay_resident macro last_instruc
    mov     dx, last_instruc
    inc     dx
    int     27H
endm
```

COMMENTS The Terminate but Stay Resident call is used to make a piece of code remain resident in the system after its termination. Typically, this call is used in .COM files to allow some device specific interrupt handler to remain resident to process asynchronous interrupts.

DX must contain the offset (from the segment address in CS) of the first byte following the last byte of code in the program. When Interrupt 27H is executed, the program terminates but is treated as an extension of MS-DOS; it remains resident and is not overlaid by other programs when it terminates.

If an executable .COM file ends with this interrupt, it becomes a resident operating system command.

This interrupt is provided for compatibility with versions of MS-DOS prior to 2.0. New programs should use Function 31H (Keep Process).

EXAMPLE

This example terminates a program, but leaves the program resident:

```
;CS must equal PSP value given at start
;(ES and DS values)
;variable last address must equal offset of last byte
;  in program
    mov dx,last_address
    inc dx
    int 27H
;There is no return from this interrupt
end
```


Functions

Most of the MS-DOS function calls require input to be passed to them in registers. After setting the proper register values, the function may be invoked in one of the following ways:

- Place the function number in AH and issue Interrupt 21H. All of the examples in this chapter use this method.
- Place the function number in AH and execute a long call to offset 50H in your Program Segment Prefix. Note that programs using this method will not operate correctly on versions of MS-DOS that are lower than 2.0.
- An additional method exists for programs that were written with different calling conventions. This method should be avoided for all new programs. The function number is placed in the CL register and other registers are set according to the function specification. Then, an intrasegment call is made to location 5 in the current code segment. Note that this method is valid only for Function Requests 01H through 24H.

REGISTERS

When MS-DOS takes control after a function call, it switches to an internal stack. Registers not used to return information (except AX) are preserved. The calling program's stack must be large enough to accommodate the interrupt system—at least 128 bytes in addition to other needs. A detailed description of each function request follows in numerical order.

00H - Terminate Program

PURPOSE Use to terminate a program.

CALL AH = 00H
 CS — Segment address of Program Segment
 Prefix

RETURN None

MACRO Terminate_program macro
 mov ah,00h
 int 21H
 endm

COMMENTS Function 00H is called by Interrupt 20H; it
 performs the same processing.

The CS register must contain the segment
address of the Program Segment Prefix before
you call this interrupt.

The following exit addresses are restored from
the specified offsets in the Program Segment
Prefix:

Exit Address	Offset
Program Terminate	0AH
Control-C	0EH
Critical Error	12H

All file buffers are flushed to disk.

Warning: *Close all files that have changed before calling this function. If a changed file is not closed, its length is not recorded correctly in the directory. See Function 10H for a description of the Close File system call.*

EXAMPLE

The following program displays a message and returns to MS-DOS:

Note: This example must have function 09H loaded to work properly.

```
message db      "Displayed by FUNC00H example", 0DH, 0AH, "$"
;
begin:  display message      ;Function 09H
        terminate_program    ;THIS FUNCTION
code    ends
        end start
```

01H - Read Keyboard and Echo

PURPOSE Use to read and echo a typed character from the keyboard. This function checks for **CTRL-C**. Use the following table to select other read functions:

Function	Check CTRL-C?	Echo?
01H	Yes	Yes
06H	No	Yes
07H	Yes	No
08H	No	No

CALL AH = 01H

RETURN AL — Character typed

MACRO

```
read_kbd_and_echo macro
    mov     ah,01H
    int     21H
endm
```

COMMENTS Function 01H waits for a character to be typed at the keyboard, then echoes the character to the display and returns it in AL. If the character is **CTRL**C, Interrupt 23H is executed.

Echoing characters to the display is by default. The output can be redirected.

Note: Extended ASCII codes must be retrieved with two calls to function 01H.

EXAMPLE

The following program displays and prints characters as they are typed: If **RETURN** is pressed, the program sends Line Feed-Carriage Return to both the display and the printer:

Note: This example must have functions 02H and 05H loaded to work properly.

```
func_01H:    read_kbd_and_echo    ;THIS FUNCTION
              print_char al       ;Function 05H
              cmp    al,0DH       ;is it a CR?
              jne    func_01H     ;if yes, also print LF
              print_char 0AH      ;Function 05H
              display_char 0AH    ;Function 02H
              jmp    func_01H     ;get another character
              end
```

02H - Display Character

PURPOSE	Use to display the character located in the DL register.
CALL	AH = 02H DL — Character to be displayed
RETURN	None
MACRO	<pre>display_char macro character mov dl,character mov ah,02H int 21H endm</pre>
COMMENTS	If CTRL C is typed, Interrupt 23H is issued.
EXAMPLE	The following program converts lowercase characters to uppercase before displaying them:

Note: This example must have function 08H loaded to work properly.

```
func_02H:    read_kbd          ;Function 08H
             cmp     al,"a"
             jl      uppercase ;don't convert
             cmp     al,"z"
             jg      uppercase ;don't convert
             sub     al,20H      ;convert to ASCII code
                                 ; for uppercase
uppercase:   display_char al;THIS FUNCTION
             jmp     func_02H: ;get another character
             end
```

03H - Auxiliary Input

PURPOSE Use to read a character from an auxiliary device. This function will wait for a character.

CALL AH = 03H

RETURN AL — Character from auxiliary device

MACRO `aux_input macro`
 `mov ah,03H`
 `int 21H`
 `endm`

COMMENTS This system call does not return a status or error code.

If **CTRL**C has been typed at console input, Interrupt 23H is issued.

EXAMPLE The following program prints characters as they are received from an auxiliary input device. It stops printing when an end-of-file character (ASCII code 26, or **CTRL-Z**) is received:

Note: This example must have function 05H loaded to work properly.

```
func_03H:      aux_input      ;THIS FUNCTION
               cmp al,1AH      ;end of file?
               je continue     ;yes, all done
               print_char al    ;Function 05H
               jmp func_03H     ;get another character

continue:      .
               .
               .
               end
```

04H - Auxiliary Output

PURPOSE Use to write the character, located in the DL register, to an auxiliary device.

CALL AH = 04H
 DL — Character for auxiliary device

RETURN None

MACRO `aux_output macro character`
 `mov dl,character`
 `mov ah,04H`
 `int 21H`
 `endm`

COMMENTS This system call does not return a status or error code.

If **CTRL**C has been typed at console input, Interrupt 23H is issued.

EXAMPLE

The following program gets a series of strings of up to 80 bytes from the keyboard, sending each to the auxiliary device. It stops when a full string (CR only) is typed:

Note: This example must have function 0AH loaded to work properly.

```

string      db      51H dup(?)          ;Function 0AH
            .
            .
            .
func_0AH:    get_string 50H,string      ;Function 0AH
            cmp      string[1]0        ;null string?
            je       continue          ;yes, all done
            mov      cx,word ptr string[1] ;get string length
            mov      bx,00H             ;set index to 0
send_it:     aux_output string[bx+2]     ;THIS FUNCTION
            inc      bx                 ;bump index
            loop     send_it            ;send another character
            jmp      func_04H           ;get another string
continue:    .
            .
            .
            end

```

05H - Print Character

PURPOSE	Use to print the character located in the DL register.
CALL	AH = 05H DL — Character for printer
RETURN	None
MACRO	<pre>print_char macro character mov dl,character mov ah,50H int 21H endm</pre>
COMMENTS	If CTRL C has been typed at console input, Interrupt 23H is issued.

EXAMPLE

The following program prints a test pattern on the printer. It stops if **CTRL**C is pressed:

```

line_number      db  0
                  .
                  .
                  .
func_05H:         mov    cx,3CH  ;print 60 lines
start-line:      mov    bl,31H  ;first printable character (!)
                  add    bl,line-num  ;to offset next character
                  push   cx      ;save number-of-lines counter
                  mov    cx,50H  ;loop counter for line
print_it:        print_char bl  ;THIS FUNCTION
                  inc     bl      ;move to next character
                  cmp     bl,7EH  ;last printable character (`)
                  jle     no_reset;not there yet
                  mov     bl,21H  ;start over with (!)
no_reset:        loop    print_it;print another character
                  print_char 13   ;carriage return
                  print_char 0AH ;line feed
                  inc     line_num;to offset 1st char. of line
                  pop     cx      ;restore number of lines counter
                  loop    start_line ;print another line
                  end

```

06H - Direct Console I/O

PURPOSE Use to read and echo a character from standard input (default is keyboard) with no check for **CTRL-C**. Execution is based on the DL register. Use the following table to select other read functions:

Function	Check CTRL-C?	Echo?
01H	Yes	Yes
06H	No	Yes
07H	Yes	No
08H	No	No

CALL AH = 06H
DL (See Text)

RETURN If Zero flag not set:
AL — Character from standard input
(with DL = FFH)

If Zero flag set:
AL — No character input
(with DL = FFH)

MACRO `dir_console_io` macro switch
 mov dl,switch
 mov ah,06H
 int 21H
 endm

COMMENTS

Processing depends on the value in DL when the function is called.

Note: Extended ASCII codes must be retrieved with two calls to Function 06H.

DL is FFH

If a character has been typed at the keyboard, it is returned in AL and the Zero flag is 0; if a character has not been typed, the Zero flag is 1.

**DL is not
FFH**

The character in DL is displayed.

This function does not check for **CTRL**C, **BREAK**, or **CTRL** **PrtSc**.

EXAMPLE

The following program sets the system clock to 0 and continuously displays the time. When any character is typed, the display stops changing; when any character is typed again, the clock is reset to 0 and the display starts again:

Note: This example must have functions 08H, 09H, 2CH, and 2DH loaded to work properly.

```
time      db "00:00:00.00",0DH,0AH,"$" ;Function 09H
ten
.
.
.
func_06H:  set_time      0,0,0,0          ;Function 2DH
read_clock: get_time      ;Function 2CH
            convert      ch,ten,time      ;end of chapter
            convert      cl,ten,time[3]   ;end of chapter
            convert      dh,ten,time[6]   ;end of chapter
            display      time             ;Function 09H
            dir_console_io 0FFH           ;THIS FUNCTION
            jne          stop             ;yes,stop timer
            jmp          read_clock       ;keep timer running
stop:      read_kbd      ;Function 08H
            jmp          func_06H         ;start over
            end
```

07H - Direct Console Input

PURPOSE Use to read and echo a character from standard input (default is keyboard) with no check for **CTRL-C**. Use the following table to select other read functions:

Function	Check CTRL-C?	Echo?
01H	Yes	Yes
06H	No	Yes
07H	Yes	No
08H	No	No

CALL AH = 07H

RETURN AL — Character from standard input

MACRO

```
dir_console_input macro
    mov     ah,07H
    int     21H
endm
```

COMMENTS This function does not echo the character or check for **CTRL**C (For a keyboard input function that echoes or checks for **CTRL**C, see Function 01H or 08H.)

Note: Extended ASCII codes must be retrieved with two calls to function 07H.

EXAMPLE

The following program prompts for a password (8 characters maximum) and places the characters into a string without echoing them:

Note: This example must have function 09H loaded to work properly.

```
func_07H:  display prompt      ;Function 09H
           mov     cx,8        ;maximum length of password
           xor     bx,bx       ; so BL can be used as index
get_pass:  dir_console_input  ;THIS FUNCTION
           cm      al,0DH      ;was it a CR?
           je      continue    ;yes all done
           mov     password[bx],al;no, put character in string
           inc     bx          ;bump index
           loop    get_pass     ;get another character
continue:  .                  ;BX has length = password+1
           .
           .
           end
```


08H - Read Keyboard

PURPOSE

Use to read a character from standard input (default is keyboard). Use the following table to select other read functions:

Function	Check CTRL-C?	Echo?
01H	Yes	Yes
06H	No	Yes
07H	Yes	No
08H	No	No

CALL

AH = 08H

RETURN

AL — Character from standard input

MACRO

```
read_kbd macro
    mov     ah,08H
    int     21H
endm
```

COMMENTS

If **CTRL**C is pressed, Interrupt 23H is executed. This function does not echo the character (for a keyboard input function that echoes the character or does not check for **CTRL**C, see Function 01H or 07H).

Note: Extended ASCII codes must be retrieved with two calls to function 08H.

EXAMPLE

The following program prompts for a password (8 characters maximum) and places the characters into a string without echoing them:

Note: This example must have function 09H loaded to work properly.

```
password      db  8 dup(?)
prompt        db  "Password: $"      ;Function 09H
.
.
.
func_08H:     display prompt          ;Function 09H
mov  cx,8     ;max length of password
xor  bx,bx    ;BL can be an index
get_pass:     read_kbd                ;THIS FUNCTION
cmp  al,0DH   ;was it a CR?
je   continue ;yes, all done
mov  password[bx],al ;no put char. in string
inc  bx       ;bump index
loop get_pass ;get another character
continue:     .                       ;BX length = password+1
.
.
end
```

09H - Display String

PURPOSE	Use to display a string starting at the offset located in the DX register. The string ends with "\$" located in the last address.
CALL	AH = 09H DS:DX — Start address offset of string to be displayed
RETURN	None
MACRO	<pre>display macro string mov dx,offset string mov ah,09H int 21H endm</pre>
COMMENTS	DX must contain the offset (from the segment address in DS) of a string that ends with "\$". The string is displayed (the \$ is not displayed).
EXAMPLE	<p>The following program displays the hexadecimal code of the key that is typed:</p> <p>Note: This example must have function 01H loaded to work properly.</p> <pre>table db "0123456789ABCDEF" sixteen db 10H result db "-00H",0DH,0AH,"\$" ;see text for \$ func_09H: read_kbd_and_echo ;Function 01H convert al,sixteen,result[3] ;end of chapter display result ;THIS FUNCTION jmp func_09H ;do it again end</pre>

0AH - Buffered Keyboard Input

PURPOSE Use to read a string of characters from the keyboard. The DX register contains the starting address of the input buffer.

CALL AH = 0AH
DS:DX — Starting address of input buffer

RETURN None

MACRO

```
get_string macro limit,buffer
    mov     dx,buffer
    mov     buffer[dx+1],limit
    mov     ah,0AH
    int     21H
endm
```

COMMENTS This function waits for characters to be typed. Characters are read from the keyboard and placed in the buffer beginning at the third byte until **RETURN** is pressed. If the buffer fills to one less than the maximum, additional characters typed are ignored and ASCII 7 (BEL) is sent to the display until **RETURN** is pressed. The string can be edited as it is being entered. If **CTRL**C is typed, Interrupt 23H is issued.

**Buffer
Contents**

DX must contain the offset (from the segment address in DS) of an input buffer in the following format:

Offset	Contents
0	Maximum Number of Characters
1	Actual Number of Characters
2	Buffer Size

Maximum Number of Characters

This byte (address) contains the value for the maximum number of characters in the buffer, including **RETURN**.

Actual Number of Characters

This byte contains the value for the actual numbers typed, not counting **RETURN**.

Buffer Size

This byte contains the value for the number of addresses in the buffer. The buffer must be at least as long as the value in **Maximum Number of Characters**.

EXAMPLE

The following program gets a 16-character maximum string from the keyboard and fills a 24-line by 80-character screen with it:

Note: This example must have function 09H loaded to work properly.

```
buffer          label byte
max_length      db      ?           ;maximum length
chars_entered   db      ?           ;number of chars.
string          db      11H dup (?) ;16 chars + CR
strings_per_line dw      0           ;how many fit on line
crlf            db      0DH,0AH,"$"
                .
                .
                .

func_OAH:        get_string 11H,buffer ;THIS FUNCTION
                xor      bx,bx      ;byte can be used as index
                mov      bl,char_entered ;get string length
                display buffer[bx+2]" $" ;Function 09H
                mov      al,50H      ;columns per line
                cbw
                div      chars_entered ;times string fits on line
                xor      ah,ah      ;clear remainder
                mov      strings_per_line,ax ;save col. counter
                mov      cx,18H      ;row counter
display_screen: push  cx            ;save it
                mov      cx,strings_per_line ;get col. counter
display_line:   display string      ;Function 09H
                loop  display_line
                display crlf         ;Function 09H
                pop     cx           ;get line counter
                loop  display_screen ;display 1 more line
                end
```

0BH - Check Keyboard Status

PURPOSE Use to check whether there are characters in the type-ahead buffer (or standard input device).

CALL AH = 0BH

RETURN AL = FFH — buffer or input device contains characters
AL = 0 — type-ahead buffer does not contain characters

MACRO `check_kbd_status macro`
 `mov ah,0BH`
 `int 21H`
 `endm`

COMMENTS If the type-ahead buffer contains characters, AL returns FFH; if not, AL returns 0. If **CTRL**C or **CTRL** **BREAK** is in the buffer, Interrupt 23H is executed.

EXAMPLE

The following program continuously displays the time until any key is pressed:

Note: This example must have functions 09H and 2CH loaded to work properly.

```
time      db  " 00:00:00.00" ,0DH,0AH," $"
ten       db   0AH
          .
          .
          .
func_OBH:  get_time                ;Function 2CH
           convert ch,ten,time      ;end of chapter
           convert cl,ten,time[3]   ;end of chapter
           convert db,ten,time[6]   ;end of chapter
           convert dl,ten,time[9]   ;end of chapter
           display time             ;Function 09H
           check_kbd_status         ;THIS FUNCTION
           cmp    al,FFH            ;has a key been typed
           je     all_done          ;yes, go home
           jmp    func_OBH          ;stop displaying time
           end
```


0CH - Flush Buffer, Read Keyboard

PURPOSE

Use to clear (empty) the keyboard type-ahead buffer and execute the function (01H, 06H, 07H, 08H, and 0AH) located in the AL register.

CALL

AH = 0CH

Function code:

AL = 01H — Read Keyboard and Echo

AL = 06H — Direct Console I/O

AL = 07H — Direct Console Input

AL = 08H — Read Keyboard

AL = 0AH — Buffered Keyboard Input

AL = Any other value — Return from function.

RETURN

AL = 0 — Type-ahead buffer was flushed;
no other processing performed.

MACRO

```
flush_and_read_kbd macro switch
    mov     al,switch
    mov     ah,0CH
    int     21H
endm
```

COMMENTS The keyboard type-ahead buffer is cleared (emptied). Further processing depends on the value in AL when the function is called.

Function 1, 6, 7, 8, or A The corresponding function is executed.

Any Other Value No further processing; AL returns 0.

EXAMPLE The following program both displays and prints characters as they are typed: If **RETURN** is pressed, the program sends Carriage Return-Line Feed to both the display and the printer.

Note: This example must have functions 02H and 05H loaded to work properly.

```
func_OCH:      flush_and_read_kbd 1      ;THIS FUNCTION
               print_char    al          ;Function 05H
               cmp           al,0DH       ;is it a CR?
               jne           func_OCH    ;no, print it
               print_char    0AH         ;Function 05H
               display_char  0AH         ;Function 02H
               print_char    0DH         ;Function 05H
               display_char  0DH         ;Function 02H
               jmp           func_OCH    ;get another character
               end
```

0DH - Reset Disk

PURPOSE	Use to write all modified buffers to disk and free all buffers in the internal cache.
CALL	AH = 0DH
RETURN	None
MACRO	<pre>reset_disk macro mov ah,0DH int 21H endm</pre>
COMMENTS	<p>Function 0DH is used to ensure that the internal buffer cache matches the disks in the drives. If buffers have been modified, but not yet written to disk, this function writes them out and marks all buffers in the internal cache as free.</p> <p>Function 0DH flushes (frees) all file buffers. It does not update directory entries; you must close files that have changed to update their directory entries (see Function 10H, Close File). This function need not be called before a disk change if all files that changed were closed. It is generally used to force a known state of the system; CTRLC interrupt handlers should call this function.</p>

EXAMPLE

The following program flushes all file buffers and selects disk A:

Note: This example must have function 0EH loaded to work properly.

```
begin: reset_disk          ;THIS FUNCTION
       select_disk "A"    ;Function 0EH
       end
```

0EH - Select Disk

PURPOSE Use to set the default drive to the value located in the DL register.

CALL AH = 0EH
DL — Drive number (0 = A, 1 = B, ...)

RETURN AL — Number of drives

MACRO

```
select_disk macro disk
    mov     dl,disk[-64]      ;ASCII offset
    mov     ah,0EH
    int     21H
endm
```

COMMENTS The drive specified in DL (0 = A, 1 = B, ...) is selected as the default drive. The number of drives is returned in AL.

EXAMPLE The following program selects the drive not currently selected in a 2-drive system:

Note: This example must have function 19H loaded to work properly.

```
func_OEH:      current_disk      ;Function 19H
               cmp     al,00H     ;drive A: selected?
               je      select_b   ;yes, select B
               select_disk "A"    ;THIS FUNCTION
               jmp     continue
select_b:      select_disk "B"    ;THIS FUNCTION
continue:     .
               .
               .
               end
```

0FH - Open File

PURPOSE	Use to open the file specified by the addressed FCB.
CALL	AH = 0FH DS:DX — Pointer to unopened FCB
RETURN	AL = 0 — Directory entry found AL = FFH — No Directory entry found FCB contents starting at address DX which contains the offset from the segment address in DS.
MACRO	<pre>open macro fcb mov dx,offset fcb mov ah,0FH int 21H endm</pre>
COMMENTS	DX must contain the offset (from the segment address in DS) of an unopened File Control Block (FCB). The disk directory is searched for the named file.

FCB
Contents

If a directory entry for the file is found, AL = 0 and the FCB, starting at the DX register offset, is as follows:

Offset	Contents
0	Drive Number
1	Name
9	Filename Extension
12	Current Block Number = 0
14	Logical Record Size = 80H (128)
16	File Size
20	Last Change Date
22	Reserved
32	Current Record Number
33	Relative Record Number

To find the starting address of an item in the FCB, add the offset shown to the value in the DX register. See "FILE CONTROL BLOCKS" in Chapter 5 for more information about the contents of an FCB.

Before performing a sequential disk operation on the file, you must set the **Current Record Number**.

Before performing a random disk operation on the file, you must set the **Relative Record Number**. If the **Logical Record Size** (128 bytes) is not correct, set it to the correct length.

Note: If a directory entry for the file is not found, AL returns FFH.

EXAMPLE

The following program prints the TEXTFILE.ASC file that is on the disk in drive B. If a partial record is in the buffer at end-of-file, the routine that prints the partial record prints characters until it encounters an end-of-file mark (ASCII code 26H, or **CTRL/Z**):

Note: This example must have functions 05H, 10H, 14H, and 1AH loaded to work properly.

```
fcb          db      2,"TEXTFILE.ASC"
             db      19H dup (?)
buffer       db      80H dup (?)
func_0FH:    set_dta buffer          ;Function 1AH
             open fcb              ;THIS FUNCTION
read_line:   read_seq fcb           ;Function 14H
             cmp      al,02H        ;end of file?
             je       all_done      ;yes, go home
             cmp      al,00H        ;more to come?
             jg       check_more    ;no, check for partial
                                     ;record
             mov      cx,80H        ;yes, print the buffer
             xor      si,si         ;set index to 0
print_it:    print_char buffer[si]  ;Function 05H
             inc      si            ;bump index
             loop     print_it      ;print next character
             jmp      read_line     ;read another record
check_more:  cmp      al,03H        ;part. record to print?
             jne      all_done      ;no
             mov      cx,80H        ;yes, print it
             xor      si,si         ;set index to 0
find_eof:    cmp      buffer[si],1AH ;end-of-file mark?
             je       all_done      ;yes
             print_char buffer[si]  ;Function 05H
             inc      si            ;bump index to next
                                     ;character
             loop     find_eof
all_done:    close fcb              ;Function 10H
             end
```


10H - Close File

PURPOSE Use to close the file specified by the addressed FCB.

CALL AH = 10H
 DS:DX — Pointer to opened FCB

RETURN AL = 0 — Directory entry found
 AL = FFH — No directory entry found

MACRO `close macro fcb`
 `mov dx,offset fcb`
 `mov ah,10H`
 `int 21H`
 `endm`

COMMENTS DX must contain the offset (to the segment address in DS) of an opened FCB. The disk directory is searched for the file named in the FCB. This function must be called after a file is changed to update the directory entry.

If a directory entry for the file is found, the location of the file is compared with the corresponding entries in the FCB. The directory entry is updated, if necessary, to match the FCB, and AL returns 0.

If a directory entry for the file is not found, AL returns FFH.

EXAMPLE

The following program checks the first byte of the MOD1.BAS file in drive B to see if it is FFH, and prints a message if it is:

Note: This example must have functions 09H, 0FH, 14H, and 1AH loaded to work properly.

```
message      db  "Not saved in ASCII format",0DH,0AH,"$"
fcb          db  2,"MOD1 BAS"
            db  19H dup (?)
buffer       db  80H dup (?)
            .
            .
            .
func_10H:    set_dta buffer          ;Function 1AH
            open   fcb              ;Function 0FH
            read_seq fcb            ;Function 14H
            cmp     buffer,FFH      ;is first byte FFH?
            jne     all_done        ;no
            display message         ;Function 09H
all_done:    close  fcb             ;THIS FUNCTION
            end
```

11H - Search for First Entry

- PURPOSE** Use to open the first matched file specified by the addressed FCB. Especially useful with wild-cards of partial filenames.
- CALL** AH = 11H
DS:DX — Pointer to unopened FCB
- RETURN** AL = 0 — Directory entry found
AL = FFH — No directory entry found

FCB contents starting at address DX
- MACRO**
- ```
search_first macro fcb
 mov dx, offset fcb
 mov ah, 11H
 int 21H
endm
```
- COMMENTS** DX must contain the offset (from the segment address in DS) of an unopened FCB. The disk directory is searched for the first matching name. The name can have the ? wild-card character to match any character. To search for hidden or system files, DX must point to the first byte of the extended FCB prefix.
- If a directory entry for the filename in the FCB is found, AL returns 0 and an unopened FCB of the same type (normal or extended) is created at the Disk Transfer Address.
- If a directory entry for the filename in the FCB is not found, AL returns FFH.

**Extended  
FCB**

If an extended FCB is used, the following search pattern is used:

**Note:** For more information see "FILE CONTROL BLOCKS" in Chapter 5.

**FILE ATTRIBUTE = 0**

If the FCB file attribute is zero, only normal file entries are found. Entries for volume label, subdirectories, hidden, and system files will not be returned.

**FILE ATTRIBUTE = HIDDEN OR SYSTEM FILES**

If the attribute field is set for hidden or system files, or directory entries, it is to be considered as an inclusive search. All normal file entries plus all entries matching the specified attributes are returned. To look at all directory entries except the volume label, the attribute byte may be set to hidden + system + directory (all 3 bits on).

**FILE ATTRIBUTE = VOLUME LABEL**

If the attribute field is set for the volume label, it is considered an exclusive search, and only the volume label entry is returned.

**EXAMPLE**

The following program verifies the existence of a file named REPORT. ASM on the disk in drive B:

**Note:** This example must have functions 09H and 1AH loaded to work properly.

```

yes db " FILE EXISTS.$"
no db " FILE DOES NOT EXIT.$"
fcb db 2," REPORT ASM"
 db 19H dup (?)
buffer db 80H dup (?)
crlf db 0DH,0AH "$"
 .
 .
 .
func_11H: set_dta ;Function 1AH
 search_first fcb ;THIS FUNCTION
 cmp al,FFH ;directory entry found?
 je not_there ;no
 display yes ;Function 09H
 jmp continue
not_there: display no ;Function 09H
continue: display crlf ;Function 09H
 .
 .
 .
end

```

## 12H - Search for Next Entry

---

|                 |                                                                                                                                                                                                                                                                  |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PURPOSE</b>  | Use after 11H function to open the next matched file specified by the addressed FCB.                                                                                                                                                                             |
| <b>CALL</b>     | AH = 12H<br>DS:DX — Pointer to unopened FCB                                                                                                                                                                                                                      |
| <b>RETURN</b>   | AL = 0 — Directory entry found<br>AL = FFH — No directory entry found<br><br>FCB contents starting at address DX which contains the offset from the segment address in DS.                                                                                       |
| <b>MACRO</b>    | <pre>search_next macro fcb     mov     dx,offset fcb     mov     ah,12H     int     21H endm</pre>                                                                                                                                                               |
| <b>COMMENTS</b> | If a directory entry for the filename in the FCB is found, AL returns 0 and an unopened FCB of the same type (normal or extended) is created at the Disk Transfer Address.<br><br>If a directory entry for the filename in the FCB is not found, AL returns FFH. |

**EXAMPLE**

The following program displays the number of files on the disk in drive B:

**Note:** This example must have functions 09H, 11H, and 1AH loaded to work properly.

```

message db "No files",0AH,0DH,"$"
files db 0
ten db 0AH
fcb db 2,"?????????"
 db 19H dup (?)
buffer db 80H dup (?)
 .
 .
 .

func_12H: set_dta buffer ;Function 1AH
 search_first fcb ;Function 11H
 je all_done ;no files on disk
 inc files ;increment counter
search_dir: search_next fcb ;THIS FUNCTION
 cmp al,FFH ;directory entry found?
 je done ;no, jump
 inc files ;increment counter
 jmp search_dir ;check again
done: convert files,ten,message ;end of chapter
all_done: display message ;Function 09H
 end

```

## 13H - Delete File

---

**PURPOSE**            Use to delete the file specified by the addressed FCB.

**CALL**                AH = 13H  
                          DS:DX — Pointer to unopened FCB

**RETURN**             AL = 0 — Directory entry found  
                          AL = FFH — No directory entry found

FCB contents starting at address DX which contains the offset from the segment address in DS.

**MACRO**              `delete macro fcb`  
                          `mov dx,offset fcb`  
                          `mov ah,13H`  
                          `int 21H`  
                          `endm`

**COMMENTS**        DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for a matching filename. The filename in the FCB can contain the wild-card character to match any character.

If a matching directory entry is found, it is deleted from the directory. If the ? wild-card character is used in the filename, all matching directory entries are deleted. AL returns 0.

If no matching directory entry is found, AL returns FFH.



**EXAMPLE**

The following program deletes each file on the disk in drive B that was last written before December 31, 1982:

**Note:** This example must have functions 09H, 11H, 12H and 1AH loaded to work properly.

```

year dw 1982
month db 12
day db 31
files db 0
message db "NO FILES DELETED.",ODH,0AH,"$"
fcb db 2,"?????????"
 db 1AH dup (?)
buffer db 80H dup (?)
;
begin: set_dta buffer ;Function 1AH
 search_first fcb ;Function 11H
 cmp al,0FFH ;directory entry found?
 jne compare ;yes
 jmp all_done ;no, no files on disk
compare: convert_date buffer ;end of chapter
 cmp cx,year ;next several lines
 jg next ;check date in directory
 cmp dl,month ; entry against date
 jg next ; above & check next file
 cmp dh,day ; if date in directory
 jge next ; entry isn't earlier.
 delete buffer ;THIS FUNCTION
 inc files ;bump deleted-files
 ; counter
next: search_next fcb ;Function 12H
 cmp al,00H ;directory entry found?
 je compare ;yes, check date
 cmp files,0 ;any files deleted?
 je all_done ;no, display NO FILES
 ; message.
 convert files,0AH,message ;end of chapter
all_done: display message ;Function 09H
 end

```

## 14H - Sequential Read

---

|                 |                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PURPOSE</b>  | Use to read the opened FCB.                                                                                                                                                                                                                                                                                                                                                           |
| <b>CALL</b>     | AH = 14H<br>DS:DX — Pointer to opened FCB                                                                                                                                                                                                                                                                                                                                             |
| <b>RETURN</b>   | AL = 00H — Read completed successfully<br>AL = 01H — EOF<br>AL = 02H — DTA too small<br>AL = 03H — EOF, partial record                                                                                                                                                                                                                                                                |
| <b>MACRO</b>    | <pre>read_seq macro fcb     mov     dx,offset fcb     mov     ah,14H     int     21H endm</pre>                                                                                                                                                                                                                                                                                       |
| <b>COMMENTS</b> | <p>DX must contain the offset (from the segment address in DS) of an opened FCB. The record pointed to by the current block (offset 0CH) and Current Record (offset 20H) fields is loaded at the Disk Transfer Address, then the Current Record and, if necessary, the Current Block fields are implemented.</p> <p>The record size is set to the value at offset 0EH in the FCB.</p> |

AL returns a code that describes the processing:

| Code | Meaning                                                                            |
|------|------------------------------------------------------------------------------------|
| 0    | Read completed successfully.                                                       |
| 1    | End-of-file, no data in the record.                                                |
| 2    | Not enough room at the Disk Transfer Address to read one record; read canceled.    |
| 3    | End-of-file; a partial record was read and padded to the record length with zeros. |

**EXAMPLE**

The following program displays the file named TEXTFILE.ASC that is on the disk in drive B; its function is similar to the MS-DOS TYPE command. If a partial record is in the buffer at end of file, the routine that displays the partial record displays characters until it encounters an end-of-file mark (ASCII 26, or CTRL-Z ).

*Note:* This example must have functions 02H, 09H, 0FH, 10H, and 1AH loaded to work properly.

```
fcbl db 2, "TEXTFILE.ASC"
 db 19H dup (?)
buffer db 80H dup (?), "$"
 .
 .
 .
func_14H: set_dta buffer ;Function 1AH
 open fcb ;Function 0FH
read_line: read_seq fcb ;THIS FUNCTION
 cmp al,01H ;end-of-file?
 je all_done ;yes
 cmp al,00H ;more to come?
 jg check_more ;no, check for partial record
 display buffer ;Function 09H
 jmp read_line ;get another record
check_more: cmp al,03H ;partial record in buffer?
 jne all_done ;no, go home
 xor si,si ;set index to 0
find_eof: cmp buffer[si],1AH ;is character EOF?
 je all_done ;yes, no more to display
 display_char buffer[si] ;Function 02H
 inc si ;bump index to next character
 jmp find_eof ;check next character
all_done: close fcb ;Function 10H
 end
```

## 15H - Sequential Write

---

|                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PURPOSE</b>  | Use to write to an opened FCB.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>CALL</b>     | AH = 15H<br>DS:DX — Pointer to opened FCB                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>RETURN</b>   | AL = 00H — Write completed successfully<br>AL = 01H — Disk full<br>AL = 02H — DTA too small                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>MACRO</b>    | <pre>write_seq macro fcb     mov     dx,offset fcb     mov     ah,15H     int     21H endm</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>COMMENTS</b> | <p>DX must contain the offset (from the segment address in DS) of an opened FCB. The record pointed to by Current Block (offset 0CH) and Current Record (offset 20H) fields is written from the Disk Transfer Address, then the fields are incremented as necessary.</p> <p>The record size is set to the value at offset 0EH in the FCB. If the Record Size is less than a sector, the data at the Disk Transfer Address is written to a buffer; the buffer is written to disk when it contains a full sector of data, or the file is closed, or a Reset Disk system call (Function 0DH) is issued.</p> |

AL returns a code that describes the processing:

| Code | Meaning                                                                                                                                                          |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0    | Transfer completed successfully                                                                                                                                  |
| 1    | Disk full; write canceled                                                                                                                                        |
| 2    | Write canceled; the area beginning at the Disk Transfer Address is too small to hold a record of data without overflowing or wrapping around a segment boundary. |

**EXAMPLE**

The following program creates a file named DIR.TMP on the disk in drive B: that contains the disk number (0 = A:, 1 = B:, etc.) and filename from each directory entry on the disk:

**Note:** This example must have functions 10H, 11H, 12H, 16H, and 1AH loaded to work properly.

```

record_size equ 0EH ;offset of Record Size
 ;field in FCB
.
.
.
fcb1 db 2,"DIR TMP"
 db 19H dup (?)
fcb2 db 2,"?????????"
 db 19H dup (?)
buffer db 80H dup (?)
.
.
func_15H: set_dta buffer ;Function 1AH
 search_first fcb2 ;Function 11H
 cmp al,0FFH ;directory entry found?
 je all_done ;no, no files on disk
 create fcb1 ;Function 16H
 mov fcb1[record_size],0CH
 ;set record size to 0CH
write_it: write_seq fcb1 ;THIS FUNCTION
 search_next fcb2 ;Function 12H
 cmp al,0FFH ;directory entry found?
 je all_done ;no, go home
 jmp write_it ;yes, write the record
all_done close fcb1 ;Function 10H
 end

```

## 16H - Create File

---

**PURPOSE** Use to create a file. The FCB data is placed under an existing entry for the file or the first empty entry.

**CALL** AH = 16H  
DS:DX — Pointer to unopened FCB

**RETURN** AL = 00H — Empty directory entry found  
AL = FFH — No empty entry directory available

**MACRO**

```
create macro fcb
 mov dx,offset fcb
 mov ah,16H
 int 21H
endm
```

**COMMENTS** DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for an empty entry or an existing entry for the specified filename.

If an empty directory entry is found, it is initialized to a zero-length file, the Open File system call (Function 0FH) is called, and AL returns 0. You can create a hidden file by using an extended FCB with the attribute byte (offset FCB-1) set to 2.

If an entry is found for the specified filename, all data in the file is released, making a zero-length file, and the Open File system call (Function 0FH) is issued for the filename (in other words, if you try to create a file that



already exists, the existing file is erased, and a new, empty file is created).

If an empty directory entry is not found and there is no entry for the specified filename, AL returns FFH.

### EXAMPLE

The following program creates a file named DIR.TMP on the disk in drive B that contains the disk number (0=A, 1=B, etc.) and filename for each directory entry on the disk:

**Note:** This example must have functions 10H, 11H, 12H, 15H, and 1AH loaded to work properly.

```

record_size equ 0EH ;offset of Record Size
 ;field of FCB
.
.
.
fcb1 db 2,"DIR TMP"
 db 19H dup (?)
fcb2 db 2,"???????????"
 db 19H dup (?)
buffer db 80H dup (?)
.
.
.
func_16H: set_dta buffer ;Function 1AH
 search_first fcb2 ;Function 11H
 cmp al,OFFH ;directory entry found?
 je all_done ;no, no files on disk
 create fcb1 ;THIS FUNCTION
 mov fcb1[record_size],0CH
 ;set record size to 12
write_it: write_seq fcb1 ;Function 15H
 search_next fcb2 ;Function 12H
 cmp al,OFFH ;directory entry found?
 je all_done ;no, go home
 jmp write_it ;yes, write the record
all_done close fcb1 ;Function 10H
 end

```

## 17H - Rename File

---

**PURPOSE** Use to rename a file. The DX register contains the starting address offset of a modified FCB which has the new name located starting at offset 11H.

**CALL** AH = 17H  
DS:DX — Pointer to Modified FCB

**RETURN** AL = 00H — Directory entry found  
AL = FFH — No directory entry found or destination already exists

**MACRO**

```
rename macro special_fcb
 mov dx,offset special_fcb
 mov ah,17H
 int 21H
endm
```

**COMMENTS** DX must contain the offset (from the segment address in DS) of an FCB with the drive number and filename filled in, followed by a second filename at offset 11H. The disk directory is searched for an entry that matches the first filename, which can contain the “?” and “\*” wild-card character.

If a matching directory entry is found, the filename in the directory entry is changed to match the second filename in the modified FCB (the two filenames cannot be the same name). If the “?” wild-card character is used in the second filename, the corresponding characters in the filename of the directory entry are not changed. AL returns 0. If the “\*” wild-card

character is used in the second filename, all characters affected by the "\*" character are not changed.

If a matching directory entry is not found or an entry is found for the second filename, AL returns FFH.

### EXAMPLE

The following program prompts for the name of a file and a new name, then renames the file:

**Note:** This example must have functions 09H, 0AH, and 29H loaded to work properly.

```
fcbl db 25H dup (?)
prompt1 db "Filename: $"
prompt2 db "New name: $"
reply db 11H dup(?)
crlf db 0DH,0AH,"$"
 .
 .
 .
func_17H: display prompt1 ;Function 09H
 get_string 0FH,reply ;Function 0AH
 display crlf ;Function 09H
 parse reply[2],fcbl ;Function 29H
 display prompt2 ;Function 09H
 get_string 0FH,reply ;Function 0AH
 display crlf ;Function 09H
 parse reply[2],fcbl[10H]
 ;Function 29H
 rename fcbl ;THIS FUNCTION
 end
```

## 19H - Get Current Disk

---

**PURPOSE**            Use to verify the current default drive number.  
The AL register contains the value.

**CALL**                AH = 19H

**RETURN**            AL — Currently selected drive  
(0=Drive A, 1=Drive B, etc.)

**MACRO**              `current_disk macro`  
                      `mov    ah,19H`  
                      `int    21H`  
                      `endm`

**COMMENTS**        None

**EXAMPLE**           The following program displays the currently  
selected (default) drive:

***Note:*** This example must have  
functions 02H and 09H loaded to work  
properly.

```
message db "Current disk is $" ;Function 09H
 ;for explanation of $
crlf db 0DH,0AH,"$"
 .
 .
 .
func_19H: display message ;Function 09H
 current disk ;THIS FUNCTION
 add al,41H ;ASCII offset
 display_char al ;Function 02H
 display_crlf ;Function 09H
 end
```

## 1AH - Set Disk Transfer Address

---

|                 |                                                                                                                                                                                                                       |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PURPOSE</b>  | Use to set the Disk Transfer Address to the value located in the DX register.                                                                                                                                         |
| <b>CALL</b>     | AH = 1AH<br>DS:DX — Disk Transfer Address                                                                                                                                                                             |
| <b>RETURN</b>   | None                                                                                                                                                                                                                  |
| <b>MACRO</b>    | <pre>set_dta macro buffer     mov     dx,offset buffer     mov     ah,1AH     int     21H endm</pre>                                                                                                                  |
| <b>COMMENTS</b> | DX must contain the offset (from the segment address in DS) of the Disk Transfer Address. Disk transfers cannot wrap around from the end of the segment to the beginning, nor can they overflow into another segment. |

**Note:** If you do not set the Disk Transfer Address, MS-DOS defaults to offset 80H in the Program Segment Prefix.

The size of the buffer that the DTA points to must be greater than or equal to the record size at open file time.

**EXAMPLE**

The following program prompts for a letter, converts the letter to its alphabetic sequence (A=1, B=2, etc.), then reads and displays the corresponding record from a file named ALPHABET.DAT on the disk in drive B. The file contains 26 records; each record is 28 bytes long.

**Note:** This example must have functions 01H, 09H, 0FH, 10H, and 21H loaded to work properly.

```
record_size equ 0EH ;offset of Record Size
 ;field of FCB
relative_record equ 21H ;offset of Relative Record
 ;field of FCB
.
.
.
fcb db 2," ALPHABETDAT"
 db 19H dup(?)
buffer db 22H dup(?)," $"
prompt db " Enter letter: $"
crlf db 0DH,0AH," $"
.
.
.
func_1AH: set_dta buffer ;THIS FUNCTION
open fcb ;Function 0FH
mov fcb[record_size],1CH ;set record size
get_char: display prompt ;Function 09H
read_kbd_and_echo ;Function 01H
cmp al,0DH ;just a CR?
je all_done ;yes, go home
sub al,41H ;convert ASCII
 ;code to record #
mov fcb[relative_record],al
 ;set relative record
display crlf ;Function 09H
read_ran fcb ;Function 21H
display buffer ;Function 09H
display crlf ;Function 09H
jmp get_char ;get next character
all_done: close fcb ;Function 10H
end
```

## 1BH - Get Default Drive Data

---

- PURPOSE**            Use to get data about the disk in the default drive.
- CALL**                AH = 1BH
- RETURN**             AL — Sectors per cluster  
                      DS:BX — Pointer to FAT ID byte  
                      CX — Bytes per sector  
                      DX — Clusters per drive
- MACRO**              `def_drive_data macro`  
                         `push  ds`  
                         `mov   al,byte ptr[bx]`  
                         `mov   ah,1BH`  
                         `int   21H`  
                         `endm`
- COMMENTS**          This function is similar to Function 36H (Get Disk Free Space), except that it returns the address of the FAT ID byte, in the BX register, instead of the number of clusters. Function 1CH (Get Drive Data) is similar except that it returns data on the disk in the default drive instead of the specified drive.

**EXAMPLE**

The following program displays a message that tells whether the default drive is a diskette or a fixed disk drive:

***Note:*** This example must have function 40H loaded to work properly.

```
stdout equ 1
;
msg db "Default drive is "
remove db "diskette."
fixed db "fixed."
crlf db 0DH,0AH
;
begin: write_handle stdout,msg,11H ;display message
 jc write_error ;not shown
 def_drive_data ;THIS FUNCTION
 cmp byte ptr[bx],0F8H ;check FAT ID
 jne diskette ;it's a diskette
 write_handle stdout,fixed,6 ;Function 40H
 jc write_error ;not shown
 jmp short all_done ;clean up, go home
diskette: write_handle stdout,remove,9;Function 40H
all_done: write_handle stdout,crlf,2 ;Function 40H
 jc write_error ;not shown
 end
```



## 1CH - Get Drive Data

---

**PURPOSE**      Use to get data about the disk in the specified drive.

**CALL**            AH = 1CH  
                     DL — Drive (0=default, 1=A, etc.)

**RETURN**        AL — Sectors per cluster  
                     (FFH if drive number is invalid)  
                     DS:BX — Pointer to FAT ID byte  
                     CX — Bytes per sector  
                     DX — Clusters per drive

**MACRO**                 `drive_data macro drive`  
                              `push  ds`  
                              `mov   dl,drive`  
                              `mov   al,byte ptr[bx]`  
                              `mov   ah,1BH`  
                              `int   21H`  
                              `pop   ds`  
                              `endm`

**COMMENTS**      The BX register returns the offset of the first byte of the FAT, which identifies the type of disk in the drive as follows:

| Value | Type of Drive                  |
|-------|--------------------------------|
| FF    | Double-sided, 8 sectors/track  |
| FE    | Single-sided, 8 sectors/track  |
| FD    | Double-sided, 9 sectors/track  |
| FC    | Single-sided, 9 sectors/track  |
| F9    | Double-sided, 15 sectors/track |
| F8    | Fixed disk                     |

The AL register returns FFH if the drive number is invalid.

This function is similar to Function 36H (Get Disk Free Space), except that it returns the address of the FAT ID byte, in the BX register, instead of the number of the available clusters. Function 1BH (Get Default Drive Data) is similar except that it returns data on the disk in the drive specified in DL instead of the default drive.

### EXAMPLE

The following program displays a message that tells whether drive B is a diskette or fixed disk drive:

**Note:** This example must have function 40H loaded to work properly.

```
stdout equ 1
;
msg db "Drive B is "
remove db "diskette."
fixed db "fixed."
crlf db 0DH,0AH
;
begin: write_handle stdout,msg,11H ;display message
 jc write_error ;not shown
 drive_data 2 ;THIS FUNCTION
 cmp byte ptr[bx],0F8H ;check FAT ID
 jne diskette ;it's a diskette
 write_handle stdout,fixed,6 ;Function 40H
 jc write_error ;not shown
 jmp short all_done ;clean up, go home
diskette: write_handle stdout,remove,9;Function 40H
all_done: write_handle stdout,crlf,2 ;Function 40H
 jc write_error ;not shown
 end
```

## 21H - Random Read

---

- PURPOSE**            Use to read the opened FCB starting at the relative record offset addressed by the starting address offset located in the DX register.
- CALL**                AH = 21H  
                         DS:DX — Pointer to opened FCB
- RETURN**             AL = 00H — Read completed successfully  
                         AL = 01H — EOF  
                         AL = 02H — DTA too small  
                         AL = 03H — EOF, partial record
- MACRO**              `read_ran macro fcb`  
                         `mov    dx,offset fcb`  
                         `mov    ah,21H`  
                         `int    21H`  
                         `endm`
- COMMENTS**         DX must contain the offset (from the segment address in DS) of an opened FCB. The Current Block (offset 0CH) and Current Record (offset 20H) fields are set to agree with the Relative Record field (offset 21H), then the record addressed by these fields is loaded at the Disk Transfer Address.

AL returns a code that describes the processing:

| Code | Meaning                                                                            |
|------|------------------------------------------------------------------------------------|
| 0    | Read completed successfully                                                        |
| 1    | End-of-file; no data in the record                                                 |
| 2    | Not enough room at the Disk Transfer Address to read one record; read canceled     |
| 3    | End-of-file; a partial record was read and padded to the record length with zeros. |

**EXAMPLE**

The following program prompts for a letter, converts the letter to its alphabetic sequence (A=1, B=2, etc.), then reads and displays the corresponding record from a file named ALPHABET.DAT on the disk in drive B. The file contains 26 records; each record is 28 bytes long.

**Note:** This example must have functions 01H, 09H, 0FH, 10H, and 1AH loaded to work properly.

```

record_size equ 0EH ;offset of Record Size
 ;field of FCB
relative_record equ 0DH ;offset of Relative Record
 ;field of FCB
.
.
.
fcb db 2," ALPHABETDAT"
 db 19H dup (?)
buffer db 22H dup(?)," $"
prompt db "Enter letter: $"
crlf db 0DH,0AH," $"
.
.
.
func_21H: set_dta buffer ;Function 1AH
 open fcb ;Function 0FH
 mov fcb[record_size],1CH;set record size
get_char: display prompt ;Function 09H
 read_kbd_and_echo ;Function 01H
 cmp al,0DH ;just a CR?
 je all_done ;yes, go home
 sub al,41H ;convert ASCII
 ;code to record #
 mov fcb[relative_record],al
 ;set relative record
 display crlf ;Function 09H
 read_ran fcb ;THIS FUNCTION
 display buffer ;Function 09H
 display crlf ;Function 09H
 jmp get_char ;get another character
all_done close fcb ;Function 10H
 end

```

## 22H - Random Write

---

**PURPOSE**            Use to write to an opened FCB starting at the relative record offset addressed by the starting address offset located in the DX register.

**CALL**                AH = 22H  
                        DS:DX — Pointer to opened FCB

**RETURN**             AL = 00H — Write completed successfully  
                        AL = 01H — Disk full  
                        AL = 02H — DTA too small

**MACRO**              `write_ran macro fcb`  
                        `mov    dx,offset fcb`  
                        `mov    ah,22H`  
                        `int    21H`  
                        `endm`

**COMMENTS**        DX must contain the offset from the segment address in DS of an opened FCB. The Current Block (offset 0CH) and Current Record (offset 20H) fields are set to agree with the Relative Record field (offset 21H), then the record addressed by these fields is written from the Disk Transfer Address. If the record size is smaller than a sector (512 bytes), the records are buffered until a sector is ready to write.

AL returns a code that describes the processing:

| Code | Meaning                                                                                                                                                       |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0    | Write completed successfully                                                                                                                                  |
| 1    | Disk is full                                                                                                                                                  |
| 2    | Write canceled; the area beginning at the Disk Transfer Area is too small to hold a record of data without overflowing or wrapping around a segment boundary. |

#### EXAMPLE

The following program prompts for a letter, converts the letter to its alphabetic sequence (A=1, B=2, etc.), then reads and displays the corresponding record from a file named ALPHABET.DAT on the disk in drive B. After displaying the record, it prompts the user to enter a changed record. If the user types a new record, it is written to the file; if the user just presses **CR**, the record is not replaced. The file contains 26 records; each record is 28 bytes long.

**Note:** This example must have functions 01H, 09H, 0FH, 10H, and 1AH loaded to work properly.

## System Calls

---

```
record_size equ 0EH ;offset of Record Size
 ;field of FCB
relative_record equ 21H ;offset of Relative Record
 ;field of FCB
.
.
.
fcb db 2,"ALPHABETDAT"
 db 19H dup (?)
buffer db 1CH dup(?),ODH,0AH,"$"
prompt1 db "Enter letter: $"
prompt2 db "New record(CR for no change): $"
crlf db 0DH,0AH,"$"
reply db 1EH dup (32)
blanks db 1CH dup (32)
.
.
.
func_22H: set_dta buffer ;Function 1AH
open fcb ;Function 0FH
mov fcb[record_size],1CH;set record size
get_char: display prompt1 ;Function 09H
read_kbd_and_echo ;Function 01H
cmp al,0DH ;just a CR?
je al,41H ;convert ASCII
 ; code to record #
mov fcb[relative_record],al
 ;set relative record
display crlf ;Function 09H
read_ran fcb ;Function 21H
display buffer ;Function 09H
display crlf ;Function 09H
display prompt2 ;Function 09H
get_string 1DH, reply ;Function 0AH
display crlf ;Function 09H
cmp reply[1],0 ;was anything typed
 ; besides cr?
je get_char ;no
 ;get another character
xor bx,bx ; to load a byte
 ; counter
move_string blanks, buffer, 1CH
 ;end of chapter
move_string reply[2], buffer, bx
 ;end of chapter
write_ran fcb ;THIS FUNCTION
jmp get_char ;get another character
all_done: close fcb ;Function 10H
end
```



## 23H - Get File Size

---

**PURPOSE**

Use to determine the size of a file.

**CALL**

AH = 23H

DS:DX — Pointer to unopened FCB

**RETURN**

AL = 00H — Directory entry found

AL = FFH — No directory entry found

**MACRO**

```
file_size macro fcb
 mov dx,offset fcb
 mov ah,23H
 int 21H
endm
```

**COMMENTS**

DX must contain the offset (from the segment address in DS) of an unopened FCB. You must set the Record Size field (offset 0EH) to the proper value before calling this function. This disk directory is searched for the first matching entry.

If a matching directory entry is found, the Relative Record field (offset 21H) is set to the number of records in the file, calculated from the total file size in the directory entry (offset 10H) and the Record Size field of the FCB (offset 0EH). AL returns 00.

If no matching directory entry is found, AL returns FFH.

**Note:** If the value of the Record Size field of the FCB (offset 0EH) doesn't match the actual number of characters in a record, this function may not return the correct file size. If the default record size (128) is not correct, you must set the Record Size field to the correct value before using this function.

**EXAMPLE**

The following program prompts for the name of a file, opens the file to set the Record Size field of the FCB to 80H, issues a File Size system call, and displays the file size and number of records in hexadecimal.

**Note:** This example must have functions 09H, 0AH, 0FH, 10H, and 29H loaded to work properly.

```

fcb db 25H dup(?)
prompt db "File name: $"
msg1 db "Record length: ",ODH,0AH," $"
msg2 db "Records: ",ODH,0AH," $"
crlf db ODH,0AH," $"
reply db 11H dup(?)
sixteen db 10H

.
.
.

func_23H: display prompt ;Function 09H
 get_string 11H,reply ;Function 0AH
 cmp reply[1],0 ;just a CR?
 jne get_length ;no, keep going
 jmp all_done ;yes, go home
get_length: display crlf ;Function 09H
 parse reply[2],fcb ;Function 29H
 open fcb ;Function 0FH
 file_size fcb ;THIS FUNCTION
 mov si,21H ;offset to Relative
 ; Record field
 mov di,9 ;reply in msg2
convert_it: cmp fcb[si],0 ;digit to convert?
 je show_it ;no, prepare message
 convert fcb[si],sixteen,msg2[di]
 inc si ;bump n-o-r index
 inc di ;bump message index
 jmp convert_it ;check for a digit
show_it: convert fcb[0EH],sixteen,msg1[1FH]
 display msg1 ;Function 09H
 display msg2 ;Function 09H
 jmp func_23H ;get a filename
all_done: close fcb ;Function 10H
 end

```

## 24H - Set Relative Record

---

|                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PURPOSE</b>  | Use to set the relative record to the same file address as the Current Block and Current Record fields.                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>CALL</b>     | AH = 24H<br>DS:DX — Pointer to opened FCB                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>RETURN</b>   | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>MACRO</b>    | <pre> set_relative_record macro fcb     mov     dx,offset fcb     mov     ah,24H     int     21H endm </pre>                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>COMMENTS</b> | DX must contain the offset (from the segment address in DS) of an opened FCB. The Relative Record field (offset 21H) is set to the same file address as the Current Block (offset 0CH) and Current Record (offset 20H) fields.                                                                                                                                                                                                                                                                   |
| <b>EXAMPLE</b>  | The following program copies a file using the Random Block Read and Random Block Write system calls. It speeds the copy by setting the record length equal to the file size and the record count to 1, and using a buffer of 32 K bytes. It positions the file pointer by setting the Current Record field (offset 20H) to 0 and using Set Relative Record to make the Relative Record field (offset 21H) point to the same record as the combination of the Current Record (offset 20H) fields. |

**Note:** This example must have functions 09H, 0FH, 0AH, 10H, 1AH, 27H, 28H, and 29H loaded to work properly.

```

current_record equ 20H ;offset of Current Record
 ;field of FCB
fsize equ 10H ;offset of File Size
 ;field of FCB
.
.
.
fcb db 25H dup (?)
filename db 11H dup(?)
prompt1 db "File to copy: $" ;Function 09H for
prompt2 db "Name of copy: $" ; explanation of $
crlf db 0DH,0AH,"$"

file_length dw ?
buffer db 32767 dup(?)
.
.
.
func_24H: set_dta buffer ;Function 1AH
 display prompt1 ;Function 09H
 get_string 0FH,filename ;Function 0AH
 display crlf ;Function 09H
 parse filename[2],fcb ;Function 29H
 open fcb ;Function 0FH
 mov fcb[current_record],0 ;Current Record
 ; field
 set_relative_record fcb ;THIS FUNCTION
 mov ax,word ptr fcb[size] ;get file size
 mov file_length,ax ;save it for
 ; ran_block_write
 ran_block_read fcb,1,ax ;Function 27H
 display prompt2 ;Function 09H
 get_string 0FH,filename ;Function 0AH
 display crlf ;Function 09H
 parse filename[2],fcb ;Function 29H
 set_relative_record fcb ;THIS FUNCTION
 mov ax,file_length ;get original file
 ; length
 ran_block_write fcb,1,ax ;Function 28H
 close fcb ;Function 10H
 end

```

## 25H - Set Interrupt Vector

---

**PURPOSE** Use to set the interrupt vector for the interrupt located in the AL register. The DX register contains the offset for the interrupt- handling routine.

**CALL** AH = 25H  
AL — Interrupt number  
DS:DX — Offset of interrupt-handling routine

**RETURN** None

**MACRO**

```
set_vector macro interrupt seg_adrs off_adrs
 push ds
 mov ax,seg_adrs
 mov ds,ax
 mov dx,off_adrs
 mov al,interrupt
 mov ah,25H
 int 21H
 pop ds
endm
```

**COMMENTS** Function 25H should be used to set a particular interrupt vector. The operating system can then manage the interrupts on a per-process basis. Note that programs should never set interrupt vectors by writing them directly in the low memory vector table.

DX must contain the offset (to the segment address in DS) of an interrupt-handling routine. AL must contain the number of the interrupt handled by the routine. The address in the vector table for the specified interrupt is set to DS:DX.

**EXAMPLE**

```
lds dx,intvector
mov ah,25H
mov al,intnumber
int 21H
;There are no errors returned
```

## 26H - Create New PSP

---

|                 |                                                                                                                                                                                                                 |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PURPOSE</b>  | Use to create a new Program Segment Prefix (PSP).                                                                                                                                                               |
| <b>CALL</b>     | AH = 26H<br>DX — Segment address of new PSP                                                                                                                                                                     |
| <b>RETURN</b>   | None                                                                                                                                                                                                            |
| <b>MACRO</b>    | <pre>create_psp macro seg_adrs     mov     dx, seg_adrs     mov     ah, 26H     int     21H endm</pre>                                                                                                          |
| <b>COMMENTS</b> | This function request has been superseded. Use Function 4B00H (Load and Execute a Program) to execute a child process unless it is imperative that your program be compatible with pre-2.11 versions of MS-DOS. |
| <b>EXAMPLE</b>  | None                                                                                                                                                                                                            |



## 27H - Random Block Read

---

- PURPOSE** Use to read a block from an opened FCB starting with the address located in the DX register. The CX register indicates the number of bytes in the block.
- CALL** AH = 27H  
CX — Number of bytes to read (block)  
DS:DX — Pointer to opened FCB
- RETURN** AL = 00H — Read completed successfully  
AL = 01H — EOF  
AL = 02H — End of segment  
AL = 03H — EOF, partial record  
CX — Number of blocks read
- MACRO**
- ```
ran_block_read macro fcb, count, rec_size
    mov     dx,offset fcb
    mov     cx,count
    mov     word ptr fcb[14],rec_size
    mov     ah,27H
    int     21H
endm
```
- COMMENTS** DX must contain the offset (to the segment address in DS) of an opened FCB. CX must contain the number of records to read; if it contains 0, the function returns without reading any records (no operation). The specified number of records, calculated from the Record Size field (offset 0EH), is read starting at the record specified by the Relative Record field (offset 21H).

The records are placed at the Disk Transfer Address.

AL returns a code that describes the processing:

Code	Meaning
0	Read completed successfully
1	End-of-file; no data in the record
2	Not enough room at the Disk Transfer Address to read one record without overflowing a segment boundary; read canceled
3	End-of-file; a partial record was read and padded to the record length with zeros.

CX returns the number of records read; the Current Block (offset 0CH), Current Record (offset 20H), and Relative Record (offset 21H) fields are set to address the next record.

EXAMPLE

The following program copies a file using the Random Block Read system call. It speeds the copy by specifying a record count of 1 and a record length equal to the file size, and using a buffer of 32K bytes; the file is read as a single record (compare to the sample program for Function 28H that specifies a record length of 1 and a record count equal to the file size).

Note: This example must have functions 09H, 0AH, 0FH, 16H, 1AH, 24H, 28H, and 29H loaded to work properly.

```

current_record equ    20H      ;offset of Current Record field
fsize          equ    10H      ;offset of File Size field
.
.
.
fcb            db        25H dup (?)
filename       db        11H dup(?)
prompt1       db        "File to copy: $" ;see Function 09H for
prompt2       db        "Name of copy: $" ;explanation of $
crlf          db        0DH,0AH,"$"

file_length   dw        ?
buffer        db        32767 dup(?)
.
.
.
func_27H:      set_dta buffer                ;Function 1AH
               display prompt1                ;Function 09H
               get_string 0FH,filename        ;Function 0AH
               display crlf                   ;Function 09H
               parse filename[2],fcb          ;Function 29H
               open   fcb                     ;Function 0FH
               mov fcb[current_record],0      ;set Current Record
               ; field
               set_relative_record fcb        ;Function 24H
               mov ax,word ptr fcb[fsize]     ;get file size
               mov  file_length,ax            ;save it for
               ; ran_block_write
               ran_block_read fcb,1,ax         ;THIS FUNCTION
               display prompt2                ;Function 09H
               get_string 0FH,filename        ;Function 0AH
               display crlf                   ;Function 09H
               parse filename[2],fcb          ;Function 29H
               create  fcb                     ;Function 16H
               mov fcb[current_record],0      ;set Current Record
               ;field
               set_relative_record fcb        ;Function 24H
               mov   ax,file_length           ;get original file
               ; size
               ran_block_write fcb,1,ax       ;Function 28H
               close  fcb
               end

```

28H - Random Block Write

PURPOSE Use to write a block to an opened FCB starting at the relative record offset addressed by the starting address offset located in the DX register. The CX register indicates the number of bytes in the block.

CALL AH = 28H
DS:DX — Pointer to opened FCB
CX — Number of bytes to write (block)
(0 = set File Size field)

RETURN AL = 00H — Write completed successfully
AL = 01H — Disk full
AL = 02H — End of segment
CX — Number of blocks written

MACRO `ran_block_write macro fcb, count, rec_size`
 `mov dx,offset fcb`
 `mov cx,count`
 `mov word ptr fcb[14],rec_size`
 `mov ah,28H`
 `int 21H`
 `endm`

COMMENTS DX must contain the offset (to the segment address in DS) of an opened FCB; CX must contain either the number of records to write or 0. The specified number of records (calculated from the Record Size field, offset 0EH) is written from the Disk Transfer Address. The records are written to the file starting at the record specified in the Relative Record field (offset 21H) of the FCB.

If CX is 0, no records are written, but the File Size field of the directory entry (offset 10H) is set to the number of records specified by the Relative Record field of the FCB (offset 21H); allocation units are allocated or released as required.

AL returns a code that describes the processing:

Code	Meaning
0	Write completed successfully
1	Disk full. No records written.
2	Not enough room at the Disk Transfer Address to write one record without overflowing a segment boundary; write canceled.

CX returns the number of records written; the current block (offset 0CH), Current Record (offset 20H), and Relative Record (offset 21H) fields are set to address the next record.

EXAMPLE

The following program copies a file using the Random Block Read and Random Block Write system calls. It copies by specifying a record count equal to the file size and a record length of 1, and using a buffer of 32K bytes; the file is copied with one disk access each to read and write (compare to the sample program of Function 27H that specifies a record count of 1 and a record length equal to file size):

Note: This example must have functions 09H, 0AH, 0FH, 10H, 16H, 1AH, 24H, 27H, and 29H loaded to work properly.

```

current_record equ 20H ;offset of Current Record field
fsize          equ 10H ;offset of File Size field
.
.
.
fcb            db 25H dup (?)
filename       db 11H dup(?)
prompt1       db "File to copy: $" ;Function 09H
prompt2       db "Name of copy: $" ;explanation of $
crlf          db 0DH,0AH,"$"
num_recs      dw ?
buffer        db 32767 dup(?)
func_28H:     set_dta buffer ;Function 1AH
              display prompt1 ;Function 09H
              get_string 0FH,filename ;Function 0AH
              display crlf ;Function 09H
              parse filename[2],fcb ;Function 29H
              open fcb ;Function 0FH
              mov fcb[current_record],0 ;set Current Record
              ; field
              set_relative_record fcb ;Function 24H
              mov ax,word ptr fcb[fsize] ;get file size
              mov num_recs, ax ;save it for
              ; ran_block_write
              ran_block_read fcb,num_recs,1; ;Function 27H
              display prompt2 ;Function 09H
              get_string 0FH,filename ;Function 0AH
              display crlf ;Function 09H
              parse filename[2],fcb ;Function 29H
              create fcb ;Function 16H
              mov fcb[current_record],0 ;set Current Record
              ; field
              set_relative_record fcb ;Function 24H
              ran_block_write fcb,num_recs,1 ;THIS FUNCTION
              close fcb ;Function 10H
              end

```

29H - Parse File Name

PURPOSE

Use to parse a filename with the string starting at the offset located in the SI register.

CALL

AH = 29H

AL — Controls parsing (see text)

DS:SI — Pointer to string to parse

ES:DI — Pointer to unopened FCB

RETURN

AL = 00H — No wild-card characters

AL = 01H — Wild-card characters used

AL = FFH — Drive letter invalid

DS:SI — Pointer to first byte past
string that was parsed

ES:DI — Pointer to unopened FCB

MACRO

```
parse macro string fcb
    mov     si,offset string
    mov     di,offset fcb
    push    es
    push    ds
    pop     es
    mov     al,0FH      ;bits 0, 1, 2, 3 on
    mov     ah,29H
    int     21H
    pop     es
endm
```

COMMENTS

SI must contain the offset (to the segment address in DS) of a string (command line) to parse; DI must contain the offset (to the segment address in ES) of an unopened FCB. The string is parsed for a filename of the form d:filename.ext; if one is found, a corresponding unopened FCB is created at ES:DI.

Bits 0-3 of AL control the parsing and processing. Bits 4-7 are ignored:

Bit	Value	Meaning
0	0	All parsing stops if a file separator is encountered.
0	1	Leading separators are ignored.
1	0	The drive number in the FCB is set to 0 (default drive) if the string does not contain a drive number.
1	1	The drive number in the FCB is not changed if the string does not contain a drive number.
2	0	The filename in the FCB is set to 8 blanks if the string does not contain a filename.
2	1	The filename in the FCB is not changed if the string does not contain a filename.

- | | | |
|---|---|--|
| 3 | 0 | The extension in the FCB is set to 3 blanks if the string does not contain an extension. |
| 3 | 1 | The extension in the FCB is not changed if the string does not contain an extension. |

If the filename or extension includes an asterisk (*), all remaining characters in the name or extension are set to question mark (?).

Filename separators:

: . ; , = + / " [] \ < > | space tab

Filename terminators include all the filename separators plus any control character. A filename cannot contain a filename terminator; if one is encountered, parsing stops.

If the string contains a valid filename:

- AL returns 1 if the filename or extension contains a wild-card character (* or ?); AL returns 0 if neither the filename nor extension contains a wild-card character.
- DS:SI points to the first character following the string that was parsed.
- ES:DI point to the first byte of the unopened FCB.

If the drive letter is invalid, AL returns FFH. If the string does not contain a valid filename, ES:DI+1 points to a blank (ASCII 20H).

EXAMPLE

The following program verifies the existence of the file named in reply to the prompt:

Note: This example must have functions 09H, 0AH, and 11H loaded to work properly.

```
fcbl           db 25H dup (?)
prompt         db "Filename: $"
reply         db 11H dup (?)
yes           db "FILE EXISTS" , 0DH,0AH," $"
no            db "FILE DOES NOT EXIST" , 0DH,0AH," $"
crl           db 0DH,0EH," $"
.
.
.
func_29H:      display      prompt      ;Function 09H
               get_string   15,reply    ;Function 0AH
               display      crlf        ;Function 09H
               parse        reply[2],fcb ;THIS FUNCTION
               search_first fcb         ;Function 11H
               cmp          al,0FFH      ;dir. entry found?
               je           not_there    ;no
               display      yes         ;Function 09H
               jmp          continue
not_there:     display      no
continue:      .
               .
               .
end
```

2AH - Get Date

PURPOSE	Use to determine the current date set in the operating system.
CALL	AH = 2AH
RETURN	AL = 0 through 6 — Day of week (0=Sunday, 1=Monday, ..., 6=Saturday) CX = 0 through 119 — Year (0=1980, 1=1981, etc.) DH = 1 through 12 — Month DL = 1 through 31 — Day
MACRO	<pre>get_date macro mov ah, 2AH int 21H endm</pre>
COMMENTS	This function returns the current date set in the operating system as binary numbers in CX and DX: CX Year (1980-2099) DH Month (1=January, 2=February, etc.) DL Day (1-31) AL Day of week (0=Sunday, 1=Monday, etc.)

EXAMPLE

The following program gets the date, increments the day, increments the month of year, if necessary, and sets the new date.

```
month      db      31,28,31,30,31,30,31,31,30,31,30,31
            .
            .
func_2AH:  get_date      ;THIS FUNCTION
            inc          dl          ;increment day
            xor          bx,bx      ;so BL can be used as index
            mov          bl,dh      ;move month to index register
            dec          bx          ;month table starts with 0
            cmp          dl,month[bx] ;past end of month?
            jle          month_ok    ;no, set the new date
            mov          dl,1        ;yes, set day to 1
            inc          dh          ;and increment month
            cmp          dh,0CH      ;past end of year?
            jle          month_ok    ;no, set the new date
            mov          dh,1        ;yes, set the month to 1
            inc          cx          ;increment year
month_ok:  set_date      cx,dh,dl    ;Function 2BH
            end
```

2BH - Set Date

PURPOSE	Use to set the current date in the operating system.
CALL	AH = 2BH CX = 0 through 119 — Year (0=1980, 1=1981, etc.) DH = 1 through 12 — Month DL = 1 through 31 — Day
RETURN	AL = 00H — Data was valid AL = FFH — Data was invalid
MACRO	<pre>set_date macro year month day mov cx,year mov dh,month mov dl,day mov ah,2BH int 21H endm</pre>
COMMENTS	Registers CX and DX must contain a valid date in binary: CX — Year (1980-2099) DH — Month (1 = January, 2 = February, etc.) DL — Day (1-31) If the date is valid, the data is set and AL returns 0. If the data is not valid, the function is canceled and AL returns FFH.

EXAMPLE

The following program gets the date, increments the day, increments the month or year, if necessary, and sets the new date:

Note: This example must have function 2AH loaded to work properly.

```
month      db          31,28,31,30,31,30,31,31,30,31,30,31

func_2BH:  get_date          ;Function 2AH
           inc              dl          ;increment day
           xor              bx,bx       ;so BL can be used as index
           mov              bl,dh       ;move month to index register
           dec              bx          ;month table starts with 0
           cmp              dl,month[bx] ;past end of month?
           jle              month_ok    ;no, set the new date
           mov              dl,1        ;yes, set day to 1
           inc              dh          ;and increment month
           cmp              dh,0CH      ;past end of year?
           jle              month_ok    ;no, set the new date
           mov              dh,1        ;yes, set the month to 1
           inc              cx          ;increment year
month_ok:   set_date         cx,dh,dl   ;THIS FUNCTION
           end
```

2CH - Get Time

PURPOSE	Use to determine the current time set in the operating system.
CALL	AH = 2CH
RETURN	CH = 0 through 23 — Hour CL = 0 through 59 — Minute DH = 0 through 59 — Seconds DL = 0 through 99 — Hundredths of a second
MACRO	<pre>get_time macro mov ah, 2CH int 21H endm</pre>
COMMENTS	This function returns the current time set in the operating system as binary numbers in CX and DX.

EXAMPLE

The following program continuously displays the time until any key is pressed:

Note: This example must have functions 09H and 0BH loaded to work properly.

```
time      db      '00:00:00.00',0DH,'$'
ten       db      0AH

        .
        .
func_2CH: get_time          ;THIS FUNCTION
          convert ch,ten,time
          convert cl,ten,time[3]
          convert dh,ten,time[6]
          convert dl,ten,time[9]
          display time      ;Function 09H
          check_kbd_status  ;Function 0BH
          cmp      al,0FFH   ;has a key been pressed?
          je       all_done  ;yes, terminate
          jmp      func_2CH  ;no, display time
all_done: end
```


2DH - Set Time

PURPOSE	Use to set the time in the operating system.
CALL	AH = 2DH CH = 0 through 23 — Hour CL = 0 through 59 — Minute DH = 0 through 59 — Seconds DL = 0 through 99 — Hundredths of a second
RETURN	AL = 00H — Time was valid AL = FFH (255) — Time was invalid
MACRO	<pre>set_time macro hour,minutes,seconds,hundredths mov ch,hour mov cl,minutes mov dh,seconds mov dl,hundredths mov ah,2DH int 21H endm</pre>
COMMENTS	Registers CX and DX must contain a valid time in binary: CH — Hour (0-23) CL — Minutes (0-59) DH — Seconds (0-59) DL — Hundredths of a second (0-99)

If the time is valid, the time is set and AL returns 0. If the time is not valid, the function is canceled and AL returns FFH.

Note: The system time must be reset following the execution of this program.

EXAMPLE

The following program acts as a stopwatch. When a character is typed, it sets the system clock to zero and begins to continuously display the time. When a second character is typed, the system stops updating the time display.

Note: This example must have functions 06H, 09H, and 2CH loaded to work properly.

```
time      db      ''00:00:00.00'',0DH,'$$'
ten       db      0AH
.
.
func_2DH:  dir_console_io 0FFH      ;Function 06H
          jz              func_2DH ;wait for keystroke
          set_time        0,0,0,0  ;THIS FUNCTION
read_clock: get_time          ;Function 2CH
          convert ch,ten,time      ;end of chapter
          convert cl,ten,time[3]   ;end of chapter
          convert dh,ten,time[6]   ;end of chapter
          convert dl,ten,time[9]   ;end of chapter
          display time            ;Function 09H
          dir_console_io 0FFH      ;Function 06H
          jz              read_clock ;no char, keep updating
          end
```

2EH - Set/Reset Verify Flag

PURPOSE Use to change the verify flag. The AL register contains the set or reset code (set=1, reset=0).

CALL AH = 2EH

 AL = 00H — Do not verify
 AL = 01H — Verify

RETURN None

MACRO `verify macro switch`
 `mov al,switch`
 `mov ah,2EH`
 `int 21H`
 `endm`

COMMENTS AL must be either 1 (verify after each disk write) or 0 (write without verifying). The system checks this flag each time it writes to disk.

The flag is normally off; you may wish to turn it on when writing critical data to disk. Because disk errors are rare and verification slows writing, you will probably want to leave it off at other times.

EXAMPLE

The following program copies the contents of a single-sided disk in drive A to the disk drive in drive B, verifying each write. It uses a buffer of 32K bytes:

Note: This example must have functions 08H, 09H, 25H, and 26H loaded to work properly.

```
on      equ    1
off     equ    0
.
.
prompt  db      'Source in A, target in B',0DH,0AH
        db      'Any key to start. $'
start   dw      0
buffer  db      40H dup (512 dup(?))          ;64 sectors
func_2EH: display prompt                      ;Function 09H
        read_kbd                             ;Function 08H
        verify on                             ;THIS FUNCTION
        mov     cx,5                          ;5 times
copy:   push    cx                            ;save counter
        abs_disk_read 0,buffer,40H,start      ;Interrupt 25H
        abs_disk_write 1,buffer,40H,start     ;Interrupt 26H
        add     start,40H                     ;do next 64 sectors
        pop     cx                            ;restore counter
        loop    copy                          ;do it again
        verify off                             ;THIS FUNCTION
end
```

2FH - Get Disk Transfer Address

PURPOSE Use to determine the current Disk Transfer Address.

CALL AH = 2FH

RETURN ES:BX — Pointer to Disk Transfer Address

MACRO

```
get_disk_xfer macro
    mov     2h,2fh
    int     21H
endm
```

COMMENTS None

EXAMPLE The following program displays the segment address of the current Disk Transfer Address in the form segment:offset:

Note: This example must have function 09H loaded to work properly.

```
message db      "DTA --      :      ",0DH,0AH,"$"
sixteen db      10H
temp      db      2 dup (?)
;
begin: get_disk_xfer                ;THIS FUNCTION
      mov     word ptr temp,ex      ;to access each byte
      convert temp[1],sixteen,message[07H] ;end of chapter
      convert temp,sixteen,message[09H]   ;end of chapter
      convert bh,sixteen,message[0CH]     ;end of chapter
      convert bl,sixteen,message[0EH]     ;end of chapter
      display message                ;Function 09H
      end
```

30H - Get MS-DOS Version Number

PURPOSE	Use to determine the current MS-DOS version.
CALL	AH = 30H
RETURN	AL — Major version number AH — Minor version number BH — OEM number BL:CX — User number (24 bits)
MACRO	<pre> get_version macro mov ah,30H int 21H endm </pre>
COMMENTS	On return, AX will contain the two-part version designation; i.e., for MS-DOS 1.28, AL would be 1 and AH would be 28. For pre-1.28 MS-DOS AL = 0. Note that version 3.1 is the same as 3.10, not the same as 3.01.
EXAMPLE	The following program displays the version of MS-DOS if it is 1.28 or later:
	<pre> message db "MS-DOS Version . ",0DH,0AH," \$" ten db 0AH ;for convert ; begin: get_version ;THIS FUNCTION cmp al,00H ;1.28 or later? jng return ;no, go home convert al,ten,message[0FH] ;end of chapter convert ah,ten,message[12H] ;end of chapter display message end </pre>

31H - Keep Process

PURPOSE Use to terminate the process and keep all allocation blocks belonging to that process.

CALL AH = 31H
 AL — Exit code
 DX — Memory size, in paragraphs

RETURN None

MACRO `keep_process macro exitcode parasize`
 `mov al,exitcode`
 `mov dx,parasize`
 `mov ah,31H`
 `int 21H`
 `endm`

COMMENTS This call terminates the current process and attempts to set the initial allocation block to a specific size in paragraphs. It will not free up any other allocation blocks belonging to that process. The exit code passed in AX is retrievable by the parent via Function 4DH.

This method is preferred over Interrupt 27H and has the advantage of allowing more than 64K bytes to be kept.

EXAMPLE None

33H - Control-C Check

PURPOSE Use this function to include **CTRL-C** checking on any system call.

CALL AH = 33H
AL = 00H — Request current state
AL = 01H — Set state

DL = 00H — Off (if setting state)
DL = 01H — On

RETURN DL = 00H — Off
DL = 01H — On

MACRO

```
ctrl_c_ck macro switch val
    mov     dl,val
    mov     al,switch
    mov     ah,33h
    int     21H
endm
```

COMMENTS MS-DOS ordinarily checks for a **CTRL-C** on the controlling device only when doing function call operations 01H-0CH to that device. Function 33H allows the user to expand this checking to include any system call. For example, with the **CTRL-C** trapping off, all disk I/O will proceed without interruption; with **CTRL-C** trapping on, the **CTRL-C** interrupt is given at the system call that initiates the disk operation.

Note: Programs that wish to use functions 06H or 07H to read **CTRL-C's** as data must ensure that the **CTRL-C** check is off.

EXAMPLE

The following program displays a message that tells whether Control-C is on or off:

Note: This example must have function 09H loaded to work properly.

```
message db      "Control-C checking ","$"
on       db      "on","$",0DH,0AH,"$"
off      db      "off","$",0DH,0AH,"$"
;
begin:  display   message           ;Function 09H
        ctrl_c_ck 0                 ;THIS FUNCTION
        cmp        dl,0             ;Is checking off?
        jg         ck_on            ;No
        display    off              ;Function 09H
        jmp        return           ;Go home
ck_on:  display    on               ;Function 09H
        end
```

35H - Get Interrupt Vector

PURPOSE	Use to determine the status of a particular interrupt vector.
CALL	AH = 35H AL — Interrupt number
RETURN	ES:BX — Pointer to interrupt routine
MACRO	<pre>get_vector macro interrupt mov al, interrupt mov ah, 35H int 21H endm</pre>
COMMENTS	This function returns the interrupt vector associated with an interrupt. Note that programs should never get an interrupt vector by reading the low memory vector table directly.

EXAMPLE

The following program displays the segment and offset (CS:IP) for the handler for Interrupt 25H (Absolute Disk Read):

Note: This example must have function 09H loaded to work properly.

```
message db      " Interrupt 25H -- CS:0000 IP:0000"
           db      0DH,0AH," $"
vec_seg db      2 dup (?)
vec_off db      2 dup (?)
;
begin: push      es           ;save ES
       get_vector 25H         ;THIS FUNCTION
       mov        ax,es       ;INT 25H segment in AX
       pop        es         ;save ES
       convert     ax,10H,message[14H];end of chapter
       convert     bx,10H,message[1CH];end of chapter
       display     message     ;Function 09H
       end
```

36H - Get Disk Free Space

PURPOSE	Use to determine the amount of free space available. Number of clusters is located in the BX register.
CALL	AH = 36H DL = 0, 1, etc. — Drive number (0=Default, 1=A, etc.)
RETURN	AL — Sectors per cluster AL = FFFFH — drive number is invalid BX — Available clusters CX — Bytes per sector DX — Clusters per drive
MACRO	<pre>get_disk_space macro drive mov di,drive mov ah,36H int 21H endm</pre>
COMMENTS	This function returns free space on a disk along with additional information about the disk.

EXAMPLE

The following program displays the space information for the disk in drive B:

Note: This example must have function 09H loaded to work properly.

```
message db "      clusters on drive B.",0DH,0AH      ;DX
        db "      clusters available.",0DH,0AH      ;BX
        db "      sectors per cluster.",0DH,0AH      ;AX
        db "      bytes per sector.",0DH,0AH,"$"      ;CX
;
begin:  get_disk_space 2                          ;THIS FUNCTION
        convert      ax,0AH,message[37H] ;end of chapter
        convert      bx,0AH,message[1CH] ;end of chapter
        convert      cx,0AH,message[53H] ;end of chapter
        convert      dx,0AH,message      ;end of chapter
        display      message              ;Function 09H
        end
```

38H - Get/Set Country Data

PURPOSE	Use to get or set the current country information. Country information is pertinent in international applications.
CALL (GET)	<p>AH = 38H</p> <p>AL = 0 — Current country AL = 1 through FEH — Country code AL = FFH — BX contains country code</p> <p>BX = 00FFH through FFFFH — Country code (When AL=FFH) DS:DX — Pointer to 32-byte memory area</p>
CALL (SET)	<p>AH = 38H</p> <p>AL = 1 through FEH — Country code AL = FFH — BX contains country code</p> <p>BX = 00FFH through FFFFH — Country code (When AL=FFH) DX = -1 (FFFFH)</p>
RETURN	<p>Carry set: AX = 2 — Invalid country code</p> <p>Carry not set: BX — Country code (get only) No Error (set only)</p>

MACROS

```
get_country macro country,buffer
    local gc_01H
    mov     dx,buffer
    mov     ax,country
    cmp     ax,FFH
    jl      gc_01H
    mov     al,FFH
    mov     bx,country
    mov     ah,38H
    int     21H
endm
```

```
set_country macro country
    local sc_01H
    mov     dx,FFFFH
    mov     ax,country
    cmp     ax,FFH
    jl      sc_01H
    mov     bx,country
    mov     al,FFH
    mov     ah,38H
    int     21H
endm
```

COMMENTS

Function 38H gets or sets the country data that MS-DOS uses to control the keyboard and display. The value passed in AL is either 0 (for current country) or a country code. Country codes are typically the international telephone prefix code for the country.

The AL register specifies the country code as follows:

AL Value	Description
0	Retrieve information about the country currently set
1 through FEH	Retrieve information about the country identified by this code.
FFH	Retrieve information about the country identified by the contents of BX.

If $DX = -1$, then the call sets the current country (as returned by the $AL = 0$ call) to the country code in AL . If the country code is not found, the current country is not changed.

Note: Applications must assume 32 bytes of information. This means the buffer pointed to by $DS:DX$ must be able to accommodate 32 bytes.

This function is fully supported only in versions of MS-DOS 2.11/3.10 and later.

This function returns, in the block of memory pointed to by $DS:DX$, information pertinent to international applications. The contents of the block are shown in the following table.

WORD Date/time format
5-BYTE ASCIIZ string currency symbol
2-BYTE ASCIIZ string thousands separator
2-BYTE ASCIIZ string decimal separator
2-BYTE ASCIIZ string date separator
2-BYTE ASCIIZ string time separator
1-BYTE Bit field
1-BYTE currency places
1-BYTE time format
4-BYTE case mapping call
2-BYTE ASCIIZ string data list separator

The format of most of the entries is ASCIIZ (a NUL terminated ASCII string), but a fixed size is allocated for each field for easy indexing into the table.

The date/time format (see table) has the following values:

0—USA standard	h:m:s m/d/y
1—Europe standard	h:m:s d/m/y
2—Japan standard	y/m/d h:m:s

The bit field contains 8-bit values. Any bit not currently defined must be assumed to have a random value.

Bit	Value	Meaning
0	0	If currency symbol precedes the currency amount
0	1	If currency symbol comes after the currency amount
1	0	If the currency symbol is directly adjacent to the currency amount.
1	1	If there is a space between the currency symbol and the amount.

The time format has the following values:

- 0 - 12 hour time
- 1 - 24 hour time

The currency places field indicates the number of places which appear after the decimal point on currency amounts.

The Case Mapping call is a FAR procedure which will perform country specific lower-to-uppercase mapping on character values from 80H to FFH. It is called with the character to be mapped in AL. It returns the correct upper case code for that character, if any, in AL. AL and the FLAGS are the only registers altered. It is allowable to pass this routine codes below 80H; however nothing is done to characters in this range. In the case where there is no mapping, AL is not altered.

EXAMPLE 1 The following program gets the time and date in the format appropriate to the current country code, and the number 999,999 and 99/100 as a currency amount with the proper currency symbol and separators:

```

time      db      " : : ",5 dup (20H)," $"
date      db      " / / ",5 dup (20H)," $"
number    db      "999?999?999",0DH,0AH," $"
data_area db      32 dup (?)
;
begin:    get_country 0,data_area ;THIS FUNCTION
          get_time    ;Function 2CH
          byte_to_dec ch,time     ;end of chapter
          byte_to_dec cl,time[03H]
          byte_to_dec dh,time[06H]
          get_date     ;Function 2AH
          sub          cx,1900    ;want last 2 digits
          byte_to_dec cl,date[06H] ;end of chapter
          cmp          word ptr data_area,0 ;check country code
          jne          not_usa    ;it's not USA
          byte_to_dec  dh,date     ;end of chapter
          byte_to_dec  dl,date[03H] ;end of chapter
          jmp          all_done    ;display data
not_usa:   byte_to_dec  dl,date     ;end of chapter
          byte_to_dec  dh,date[03H] ;end of chapter
all_done:  mov          al,data_area[07H] ;Thousand separator
          mov          number[03H],al   ;put in NUMBER
          mov          al,data_area[09H] ;decimal separator
          mov          number[07H],al   ;put in AMOUNT
          display      time             ;Function 09H
          display      date             ;Function 09H
          display_char data_area[02H]  ;Function 02H
          display      number          ;Function 09H
          end

```

EXAMPLE 2 The following program sets the country code to the United Kingdom (44H).

```

uk      equ      2CH
;
begin:  set_country uk ;THIS FUNCTION
        jc          error ;not shown
        end

```

39H - Create Directory

PURPOSE	Use to add sub-directories under current directory.
CALL	AH = 39H DS:DX — Pointer to pathname
RETURN	Carry set: AX = 3 — Path not found AX = 5 — Access denied Carry not set: AX — No error
MACRO	<pre>make_dir macro name mov dx,offset name mov ah,39h int 21H endm</pre>
COMMENTS	Given a pointer to an ASCIIZ name, this function creates a new directory entry at the end.

EXAMPLE

The following program adds a subdirectory named NEWDIR to the root directory on the disk in drive B, changes the current directory to NEWDIR, changes the current directory back to the original directory, then deletes NEWDIR. It displays the current directory after each step to confirm the changes:

Note: This example must have functions 3AH, 3BH, and 47H loaded to work properly.

```

old_path      db      "b:\",0,3FH dup (?)
new_path      db      "b:\new_dir",0
buffer        db      "b:\",0,3FH dup (?)
;
begin: get_dir      2,old_path[03H]          ;Function 47H
      jc          error_get                  ;not shown
      display_asciiz old_path                ;end of chapter
      make_dir     new_path                  ;THIS FUNCTION
      jc          error_make                 ;not shown
      change_dir   new_path                  ;Function 3BH
      jc          error_change               ;not shown
      get_dir      2,buffer[03H]             ;Function 47H
      jc          error_get                  ;not shown
      display_asciiz buffer                  ;end of chapter
      change_dir   old_path                  ;Function 3BH
      jc          error_change               ;not shown
      rem_dir      new_path                  ;Function 3AH
      jc          error_rem                  ;not shown
      get_dir      2,buffer[03H]             ;Function 47H
      jc          error_get                  ;not shown
      display_asciiz buffer                  ;end of chapter
      end

```

3AH - Remove Directory

PURPOSE	Use to remove sub-directories under current directory.
CALL	AH = 3AH DS:DX — Pointer to pathname
RETURN	Carry set: AX = 3 — Path not found AX = 5 — Access denied AX = 16 — Current directory Carry not set: AX — No error
MACRO	<pre>rem_dir macro name mov dx,offset name mov ah,3AH int 21H endm</pre>
COMMENTS	Function 3AH is given an ASCIIZ name of a directory. That directory is removed from its parent directory.

EXAMPLE

The following program adds a subdirectory named NEWDIR to the root directory on the disk in drive B, changes the current directory to NEWDIR, changes the current directory back to the original directory, then deletes NEWDIR. It displays the current directory after each step to confirm the changes:

Note: This example must have functions 39H, 3BH, and 47H loaded to work properly.

```

old_path      db      "b:\",0,3FH dup (?)
new_path      db      "b:\new_dir",0
buffer        db      "b:\",0,3FH dup (?)
;
begin: get_dir      2,old_path[03H]          ;Function 47H
      jc          error_get                  ;not shown
      display_asciiz old_path                ;end of chapter
      make_dir     new_path                  ;Function 39H
      jc          error_make                 ;not shown
      change_dir   new_path                  ;Function 3BH
      jc          error_change               ;not shown
      get_dir      2,buffer[03H]             ;Function 47H
      jc          error_get                  ;not shown
      display_asciiz buffer                  ;end of chapter
      change_dir   old_path                  ;Function 3BH
      jc          error_change               ;not shown
      rem_dir      new_path                  ;THIS FUNCTION
      jc          error_rem                  ;not shown
      get_dir      2,buffer[03H]             ;Function 47H
      jc          error_get                  ;not shown
      display_asciiz buffer                  ;end of chapter
      end

```

3BH - Change the Current Directory

PURPOSE Use to change the current directory.

CALL AH = 3BH
 DS:DX — Pointer to pathname

RETURN Carry set:
 AX = 3 — Path not found
 Carry not set:
 AX — No error

MACRO `change_dir macro name`
 `mov dx,offset name`
 `mov ah,3BH`
 `int 21H`
 `endm`

COMMENTS Function 3BH is given the ASCIIZ name of the directory which is to become the current directory. If any member of the specified pathname does not exist, then the current directory is unchanged. Otherwise, the current directory is set to the string.

EXAMPLE

The following program adds a subdirectory named NEWDIR to the root directory on the disk in drive B, changes the current directory to NEWDIR, changes the current directory back to the original directory, then deletes NEWDIR. It displays the current directory after each step to confirm the changes:

Note: This example must have functions 39H, 3AH and 47H loaded to work properly.

```

old_path      db      "b:\",0,3FH dup (?)
new_path      db      "b:\new_dir",0
buffer        db      "b:\",0,3FH dup (?)
;
begin: get_dir  2,old_path[03H]          ;Function 47H
      jc      error_get                  ;not shown
      display_asciiz old_path            ;end of chapter
      make_dir new_path                  ;Function 39H
      jc      error_make                  ;not shown
      change_dir new_path                ;THIS FUNCTION
      jc      error_change                ;not shown
      get_dir  2,buffer[03H]             ;Function 47H
      jc      error_get                  ;not shown
      display_asciiz buffer              ;end of chapter
      change_dir old_path                ;THIS FUNCTION
      jc      error_change                ;not shown
      rem_dir  new_path                  ;Function 3AH
      jc      error_rem                  ;not shown
      get_dir  2,buffer[03H]             ;Function 47H
      jc      error_get                  ;not shown
      display_asciiz buffer              ;end of chapter
      end

```

3CH - Create Handle

PURPOSE	Use to add a handle to the current directory.
CALL	AH = 3CH CX — File attribute DS:DX — Pointer to pathname
RETURN	Carry set: AX = 3 — Path not found AX = 4 — Too many open files AX = 5 — Access denied Carry not set: AX — Handle number
MACRO	<pre> create_handle macro name attrib mov dx,offset name mov cx,attrib mov ah,3CH int 21H endm </pre>
COMMENTS	This function creates a new file or truncates an old file to zero length, in preparation for writing. DX must point to the ASCIIZ path to the file. If the file does not exist, the file is created in the appropriate directory with the attributes specified in CX. Also, the file handle returned has been opened for read/write access.

EXAMPLE

The following program creates a file named DIR.TMP on the disk in drive B, that contains the filename of each file in the current directory:

Note: This example must have functions 1AH, 3EH, 40H, 4EH, and 09H loaded to work properly.

```

srch_file      db      ":b:*.\"",0
tmp_file       db      "b:dir.tmp",0
buffer         db      26H dup (?)
handle         dw      ?
;
begin:         set_dta      buffer                ;Function 1AH
               find_first_file srch_file,16H;Function 4EH
               cmp         ax,12H                ;directory empty?
               je          all_done              ;yes, go home
               create_handle tmp_file,0          ;THIS FUNCTION
               jc          error                 ;not shown
               mov         handle,ax             ;save handle
write_it:      write_handle handle,buffer[1EH],12H
               find_next_file                    ;Function 4FH
               cmp         ax,12H                ;another entry?
               je          all_done              ;no, go home
               jmp         write_it             ;yes, write record
all_done:      close_handle handle              ;Function 3EH
               jc          error_close          ;not shown
               end

```

3DH - Open Handle

PURPOSE	Use to associate a handle with a file.								
CALL	AH = 3DH Access: AL = 0 — File opened for reading AL = 1 — File opened for writing AL = 2 — File opened for both reading and writing DS:DX — Pointer to filename								
RETURN	Carry set: AX = 2 — File not found AX = 4 — Too many open files AX = 5 — Access denied AX = CH — Invalid access Carry not set: AX — Handle number								
MACRO	<pre>open_handle macro lds dx,offset name mov ah,3DH mov al,access int 21H</pre>								
COMMENTS	Function 3DH associates a 16-bit handle with a file. The following values are allowed in AL: <table><thead><tr><th>Access</th><th>Function</th></tr></thead><tbody><tr><td>0</td><td>Opened for reading</td></tr><tr><td>1</td><td>Opened for writing</td></tr><tr><td>2</td><td>Opened for both reading and writing.</td></tr></tbody></table>	Access	Function	0	Opened for reading	1	Opened for writing	2	Opened for both reading and writing.
Access	Function								
0	Opened for reading								
1	Opened for writing								
2	Opened for both reading and writing.								

DS:DX points to an ASCIIZ name of the file to be opened.

The read/write pointer is set at the first byte of the file and the record size of the file is 1 byte. The returned file handle must be used for subsequent I/O to the file.

EXAMPLE

The following program prints the file named TEXTFILE.ASC on the disk in drive B:

Note: This example must have function 05H loaded to work properly.

```
file      db  "b:textfile.asc",0
buffer    db  ?
handle     db  ?
;
begin:     open_handle  file,0           ;THIS FUNCTION
mov        handle,ax      ;save handle
read_char: read_handle  handle,buffer,1 ;read 1 char
jc         error_read     ;not shown
cmp        ax,0           ;end of file?
je         return         ;Yes, go home
print_char buffer         ;Function 05H
jmp        read_char      ;read another
end
```

3EH - Close Handle

PURPOSE Use to close a file and clear internal buffers.

CALL AH = 3EH
 BX — File handle

RETURN Carry set:
 AX = 6 — Invalid handle

 Carry not set:
 AX not= 6 — No error

MACRO `close_handle macro handle`
 `mov bx,handle`
 `mov ah,3EH`
 `int 21H`
 `endm`

COMMENTS If BX is passed a file handle (like that returned by Functions 3DH, 3CH, or 45H), Function 3EH closes the associated file. Internal buffers are cleared.

EXAMPLE

The following program creates a file named DIR.TMP on the disk in drive B, that contains the filename of each file in the current directory:

Note: This example must have functions 1AH, 3CH, 3EH, 40H, 4EH, and 4FH loaded to work properly.

```

srch_file      db      ":b:*.\"",0
tmp_file       db      "b:dir.tmp",0
buffer         db      26H dup (?)
handle         dw      ?
;
begin: set_dta   buffer           ;Function 1AH
      find_first_file srch_file,16H;Function 4EH
      cmp          ax,12H         ;directory empty?
      je           all_done       ;yes, go home
      create_handle tmp_file,0     ;Function 3CH
      jc           error          ;not shown
      mov          handle,ax       ;save handle
write_it: write_handle handle,buffer[1EH],12H ;Function 40H
      find_next_file           ;Function 4FH
      cmp          ax,12H         ;another entry?
      je           all_done       ;no, go home
      jmp          write_it       ;yes, write record
all_done: close_handle handle      ;THIS FUNCTION
      jc           error_close    ;not shown
      end

```

3FH - Read Handle

PURPOSE	Use to read from a file to a buffer location.
CALL	AH = 3FH BX — File handle CX — Bytes to read DS:DX — Pointer to buffer
RETURN	Carry set: AX = 5 — Error set AX = 6 — Invalid handle AX = Any other value — Number of bytes read Carry not set: AX — Number of bytes read
MACRO	<pre>read_handle macro buffer,count,handle lds dx,offset buffer mov cx,count mov bx,handle mov ah,3FH int 21H endm</pre>
COMMENTS	Function 3FH transfers count bytes from a file into a buffer location. It is not guaranteed that all count bytes will be read; for example, reading from the keyboard will read at most one line of text. If the returned value is zero, then the program has tried to read from the end of file. All I/O is done using normalized pointers; no segment wraparound will occur.

EXAMPLE

The following program displays the file named TEXTFILE.ASC on the disk in drive B:

Note: This example must have functions 09H and 3DH loaded to work properly.

```

filename  db  "b:\textfile.asc",0
buffer    db  81H dup (?)
handle    dw  ?
;
begin:    open_handle filename,0      ;Function 3DH
          jc      error_open         ;not shown
          mov     handle,ax          ;save handle
read_file: read_handle buffer,file_handle,80H ;THIS FUNCTION
          jc      error_open         ;not shown
          cmp     ax,0               ;end of file?
          je      return             ;yes, go home
          mov     bx,ax              ;# of bytes read
          mov     buffer [bx],"$"    ;make a string
          display buffer             ;Function 09H
          jmp     read_file          ;read more
          end

```

40H - Write Handle

PURPOSE	Use to write to a file from a buffer location.
CALL	AH = 40H BX — File handle CX — Bytes to write DS:DX — Pointer to buffer
RETURN	Carry set: AX = 5 — Access denied AX = 6 — Invalid handle AX = Any other value — Number of bytes written Carry not set: AX — Number of bytes written
MACRO	<pre>write_handle macro buffer,count lds dx,buffer mov cx,count mov ah,40H int 21H endm</pre>
COMMENTS	<p>Function 40H transfers count bytes from a buffer into a file. It should be regarded as an error if the number of bytes written is not the same as the number requested.</p> <p>The write system call with a count of zero (CX = 0) will set the file size to the current position. Allocation units are allocated or released as required. All I/O is done using normalized pointers; no segment wraparound will occur.</p>

EXAMPLE

The following program creates a file named DIR.TMP on the disk in drive B, that contains the filename of each file in the current directory:

Note: This example must have functions 1AH, 3CH, 3EH, 4EH, and 4FH loaded to work properly.

```

srch_file      db      ":b:*.*",0
tmp_file       db      "b:dir.tmp",0
buffer         , db      26H dup (?)
handle         dw      ?
;
begin:  set_dta      buffer           ;Function 1AH
        find_first_file srch_file,16H ;Function 4EH
        cmp         ax,12H           ;directory empty?
        je          all_done         ;yes, go home
        create_handle tmp_file,0      ;Function 3CH
        jc          error            ;not shown
        mov         handle,ax        ;save handle
write_it: write_handle handle,buffer[1EH],12H ;THIS FUNCTION
        find_next_file              ;Function 4FH
        cmp         ax,12H           ;another entry?
        je          all_done         ;no, go home
        jmp         write_it         ;yes, write record
all_done: close_handle handle         ;Function 3EH
        jc          error_close      ;not shown
        end

```

41H - Delete a Directory Entry

PURPOSE	Use to remove a sub-directory from a directory.
CALL	AH = 41H DS:DX — Pointer to pathname
RETURN	Carry set: AX = 2 — File not found AX = 5 — Access denied Carry not set: No error
MACRO	<pre>delete_entry macro name lds dx,offset name mov ah,41H int 21H</pre>
COMMENTS	None

EXAMPLE

The following program deletes all files on the disk in drive B whose date is earlier than December 31, 1981:

Note: This example must have functions 0EH, 09H, 1AH, and 4EH loaded to work properly.

```

year      db      1981
month     db      12
day       db      31
files     db      ?
ten       db      0AH
message   db      "NO FILES DELETED.",0DH,0AH,"$"
path      db      "b:*. *",0
buffer    db      26H dup (?)
;
begin:    set_dta    buffer           ;Function 1AH
          select_disk "B"           ;Function 0EH
          find_first_file path,0     ;Function 4EH
          jc         all_done        ;go home if empty
compare:  convert_date buffer       ;end of chapter
          cmp        cx,year         ;after 1981?
          jg         next            ;yes, don't delete
          cmp        dl,month        ;after December?
          jg         next            ;yes, don't delete
          cmp        dh,day          ;31st or after?
          jge        next            ;yes, don't delete
          delete_entry buffer[1EH]   ;THIS FUNCTION
          jc         error_delete    ;not shown
          inc        files           ;bump file counter
next:     find_next_file             ;check directory
          jnc        compare         ;go home if done
how_many: cmp        files,0         ;was directory empty?
          je         all_done        ;yes, go home
          convert    files,ten,message ;end of chapter
all_done: display    message         ;Function 09H
          select_disk "A"           ;Function 0EH
          end

```

42H - Move File Pointer

PURPOSE	Use to move a file pointer for read/write purposes.
CALL	AH = 42H Offset — Distance to move, in bytes: CX — Upper 16 bits of offset DX — Lower 16 bits of offset Method of moving: AL = 0 — Pointer is moved to beginning of file plus offset AL = 1 — Pointer is moved to current location plus offset AL = 2 — Pointer is moved to end of file plus offset BX — File handle
RETURN	Carry set: AX = 1 — Invalid function AX = 6 — Invalid handle Carry not set: New Pointer Location: DX — Upper 16 bits of pointer AX — Lower 16 bits of pointer

MACRO

```
move_ptr macro offsetlow,offsethigh,method,  
             handle  
  
    mov  dx,offsetlow  
    mov  cx,offsethigh  
    mov  al,method  
    mov  bx,handle  
    mov  ah,42H  
    int  21H  
    endm
```

COMMENTS

Offset should be regarded as a 32-bit integer with CX occupying the most significant 16 bits and DX the least.

Function 42H moves the read/write pointer according to one of the following methods:

Method 0

The pointer is moved to offset bytes from the beginning of the file.

Method 1

The pointer is moved to the current location plus offset.

Method 2

The pointer is moved to the end of file plus offset.

EXAMPLE

The following program prompts for a letter, converts the letter to it's alphabetical sequence (A=1, B=2, etc.), then reads and displays the corresponding record from the file named ALPHABET.DAT in the current directory on the disk in drive B. The file contains 26 records of 28 bytes each:

Note: This example must have functions 01H, 09H, and 3DH loaded to work properly.

```
file      db      "b:alphabet.dat",0
buffer    db      1CH dup (?)," $"
prompt    db      "Enter letter: $"
crlf      db      0DH,0AH," $"
handle    db      ?
record_length dw 1CH
;
begin:    open_handle file      ;Function 3DH
          jc      error_open    ;not shown
          mov     handle,ax      ;save handle
get_char: display prompt        ;Function 09H
          read_kbd_and_echo      ;Function 01H
          sub     al,41h         ;convert to sequence
          mul     byte ptr record_length ;calculate offset
          move_ptr handle,0,ax,0 ;THIS FUNCTION
          jc      error_move     ;not shown
          read_handle handle,buffer,record_length
          jc      error_read      ;not shown
          cmp     ax,0            ;end of file?
          je      return         ;yes, go home
          display crlf           ;Function 09H
          display buffer         ;Function 09H
          display crlf           ;Function 09H
          jmp     get_char        ;get another character
end
```


43H - Get/Set Attributes

PURPOSE

Use to determine or set the attributes of a file via the CX register.

CALL

AH = 43H

DS:DX — Pointer to pathname

Function Code:

AL = 0 — Return in CX

AL = 1 — Set to CX

CX — Attribute to be set when AL = 1

RETURN

Carry set:

AX = 1 — Invalid function

AX = 3 — Path not found

AX = 5 — Access denied

Carry not set:

CX — Attributes when AL = 0

MACRO

```
change_attr macro name,attribute,function
    mov  dx,offset name
    mov  cx,attribute
    mov  al,function
    int  ah,43H
    int  21H
endm
```

COMMENTS

Given an ASCII filename, Function 43H will either set/get the attributes of the file to those given in CX or set the attributes of the file in CX, depending on the value of AL.

Function 0 This function will set the attributes of the file in CX.

Function 1 This function will set/get the attributes of the file to those given in CX.

EXAMPLE The following program displays the attributes assigned to the file named REPORT.ASM in the current directory on the disk in drive B:

Note: This example must have functions 02H and 09H loaded to work properly.

```
header    db    0FH dup (20h), "Read-", 0DH, 0AH
           db    "Filename      Only      Hidden "
           db    "System      Volume  Sub-Dir  Archive"
           db    0DH, 0AH, 0DH, 0AH, "$"
path       db    "b:report.asm", 3 dup (0), "$"
attribute  dw    ?
blanks     db    9 dup (20h), "$"
;
begin:     change_attr path, 0, 0 ;THIS FUNCTION
           jc      error_mode     ;not shown
           mov     attribute, cx   ;save attribute byte
           display header          ;Function 09H
           display path            ;Function 09H
           mov     cx, 6           ;check 6 bits (0-5)
           mov     bx, 1           ;start with bit 0
chk_bit:   test    attribute, bx   ;is the bit set?
           jz      no_attr         ;no
           display_char "X"        ;Function 02H
           jmp short next_bit      ;done with this bit
no_attr:   display_char 20h        ;Function 02H
next_bit:  display blanks          ;Function 09H
           shl     bx, 1           ;move to next bit
           loop    chk_bit         ;check it
           end
```

4400H, 4401H - IOCTL Data

PURPOSE Use to get/set the I/O control data for a device.

CALL AH = 44H

 AL = 00H — Get device data
 AL = 01H — Set device data

 BX — Handle
 DX — Device data

RETURN Carry set:
 AX = 1 — Invalid function
 AX = 6 — Invalid handle

 Carry not set:
 DX — Device data

MACRO `ioctl_data macro function,handle,buffer`
 `mov al,function`
 `mov ah,44H`
 `mov bx,handle`
 `mov dx,offset buffer`
 `int 21H`
 `endm`

COMMENTS This function gets/sets device information and returns it in DX. The BX register must contain the handle of a character device, such as a printer or serial port. If AL is "1", the DH register must contain "0". The device information is contained in DX in this format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cnts	R E S V	C T R L	RESERVED						D E V	E O F	R A W	R E S V	C L K	N U L	C O T	C I N

DEV - BIT 7

This bit indicates whether the channel is a device or disk file. The bit is set (=1) for a device and clear (=0) for a disk file.

For DEV = 1:

EOF = 0 — for End of File on input

RAW = 1 — Device is in Raw mode

RAW = 0 — Device is cooked

CLK = 1 — Device is the clock device

NUL = 1 — Device is the null device

COT = 1 — Device is the console output

CIN = 1 — Device is the console input

CTRL = 0 — Device cannot do control strings
when AL = 2 or 3.

CTRL = 1 — Device can process control strings
when AL = 2 or 3.

Note: This bit cannot be set (read only).

For DEV = 0:

The device information is contained in DX in this format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cnts	R E S V	C T R L	RESERVED						0	E O F	DEVICE #					

EOF = 0 — When channel has been written.

DEVICE # — The block device number for the channel (0=A, 1=B,...).

EXAMPLE

This program gets the device data for Standard Output and sets the bit that specifies not to check for control characters (bit 5), then clears the bit.

```

get      equ      0
set      equ      1
stdout   equ      1
;
begin:   ioctl_data   get,stdout   ;THIS FUNCTION
        jc           error        ;routine not shown
        mov          dh,0         ;clear DH
        or           dl,20H       ;set set bit 5
        ioctl_data   set,stdout   ;THIS FUNCTION
        jc           error        ;routine not shown
;
;   <control characters now treated as data, or 'raw mode'>
;
        ioctl_data   get,stdout   ;THIS FUNCTION
        jc           error        ;routine now shown
        mov          dh,0         ;clear DH
        and          dl,ODFH      ;clear bit 5
        ioctl_data   set,stdout   ;THIS FUNCTION
;
;   <control characters now interpreted, or 'cooked mode'>

```

4402H, 4403H - IOCTL Character

PURPOSE Use to read/write the I/O control for a character device.

CALL AH = 44H

AL = 02H — Receive control data
AL = 03H — Send control data

BX — Handle
CX — Number of bytes (count)
DS:DX — Pointer to buffer

RETURN Carry set:

AX = 1 — Invalid function
AX = 6 — Invalid handle

Carry not set:
AX — Bytes transferred

MACRO `ioctl_char macro function,handle,count,pointer`

```
mov    al,function
mov    ah,44H
mov    bx,handle
mov    cx,count
mov    dx,offset pointer
int    21H
endm
```

COMMENTS These functions allow arbitrary control strings to be read or written to or from a character device.

An **invalid function** error is returned if the CTRL bit (14) (see Function 4400H) in DX is "0".

Note: The device driver must be written to support the IOCTL interface.

EXAMPLE None

4404H, 4405H - IOCTL Block

PURPOSE Use to read/write the I/O control for a drive.

CALL AH = 44H

AL = 04H — Read bytes into DX from drive

AL = 05H — Write bytes from DX to drive

BL — Drive (0=Default,1=A,2=B,...)

CX — Number of Bytes (count)

DS:DX — Pointer to buffer

RETURN

Carry set:

AX = 1 — Invalid function

AX = 5 — Invalid drive

Carry not set:

AX — Bytes transferred

MACRO

```
ioctl_block macro function,drive,count,pointer
    mov     al,function
    mov     ah,44H
    mov     bl,drive
    mov     bx,count
    mov     dx,offset pointer
    int     21H
endm
```

COMMENTS

These functions allow arbitrary control strings to be read or written to or from a drive.

An **invalid function** error is returned if the CTRL bit (14) (see Function 4400H) in DX is 0.

Note: The device driver must be written to support the IOCTL interface. To determine this, use Function 4400H to get the device data and test bit 14; If it is set (=1), the driver supports IOCTL.

An **access denied** error is returned by, if the drive number is invalid.

EXAMPLE

None

4406H, 4407H - IOCTL Status

PURPOSE Use to get the I/O control input or output status for a device.

CALL AH = 44H

 AL = 06H — Get input status
 AL = 07H — Get output status

 BX — Handle

RETURN Carry set:

 AX = 1 — Invalid function
 AX = 5 — Access denied
 AX = 6 — Invalid handle
 AX = 13 — Invalid data

 Carry not set:

 AL = 00H — Not ready
 AL = FFH — Ready

MACRO `ioctl_status macro function,handle`

`mov al,function`
 `mov ah,44H`
 `mov bx,handle`
 `int 21H`
 `endm`

COMMENTS These two functions allow the user to check if a file handle is ready for input or output. Status of handles open to a device is the intended use of these functions, but status of a handle open to a disk file is allowed, and is defined as follows:

Value	Meaning for Device	Meaning for Input File	Meaning for Output File
00H	Not ready	Pointer at EOF	Ready
FFH	Ready	Ready	Ready

The status is defined at the time the function is CALLED. On future versions, by the time control is returned to the user from the system, the status returned may NOT correctly reflect the true current state of the device or file.

Note: An output file always returns "ready", even if the disk is full.

EXAMPLE

The following program displays a message that tells whether the file associated with handle 6 is ready for input or at end-of-file:

Note: This example must have function 40H loaded to work properly.

```

stdout          equ          1
;
message         db           "File is "
ready           db           "ready."
at_eof          db           "at EOF."
crlf            db           ODH, OAH
;
begin: write_handle stdout, message, 8      ;display message
      jc write_error                      ;not shown
      ioctl_status 6                      ;THIS FUNCTION
      jc ioctl_error                      ;not shown
      cmp al, 0                          ;check status code
      jne not_eof                          ;file is ready
      write_handle stdout, at_eof, 7       ;Function 40H
      jc write_error                      ;not shown
      jmp all_done                        ;clean up, go home
not_eof: write_handle stdout, ready, 6     ;Function 40H
all_done: write_handle stdout, crlf, 2     ;Function 40H
      jc write_error                      ;not shown
      end

```

4408H - IOCTL is Changeable

PURPOSE	Use to determine whether the drive contains a fixed or removable disk,
CALL	AX = 4408H — Get whether fixed or removable BL — Drive (0=Default,1=A,2=B,...)
RETURN	Carry set: AX = 1 — Invalid function AX = 15 — Invalid drive Carry not set: AX = 0 — Changeable AX = 1 — Not changeable
MACRO	<pre>ioctl_change macro drive mov ax,4408H mov bl,drive int 21H endm</pre>
COMMENTS	<p>This function checks whether a drive contains a fixed or removable disk. The BL register must contain the drive number. The AX register contains "0" if the disk can be changed (removable), "1" if the drive cannot.</p> <p>This function lets a program determine whether to display a message to change disks.</p>

EXAMPLE

The following program checks whether the current drive contains a removable disk. If not, processing continues; if so, it prompts the user to replace the disk in the current drive:

Note: This example must have functions 02H, 19H, and 40H loaded to work properly.

```

stdout equ 1
;
message db "Please replace disk in drive"
drives db "ABCD"
crlf db 0DH,0AH
;
begin: ioctl_change 0 ;THIS FUNCTION
      jc ioctl_error ;not shown
      cmp ax,0 ;current drive changeable?
      jne continue ;no, continue processing
      write_handle stdout,message,1DH ;Function 40H
      jc write_error ;not shown
      current_disk ;Function 19H
      xor bx,bx ;clear index
      mov bl,al ;get current drive
      display_char drives [bx] ;Function 02H
      write_handle stdout,crlf,2 ;Function 40H
      jc write_error ;not shown
continue:
;      (** FURTHER PROCESSING HERE **)

```

4409H - IOCTL is Redirected Block

PURPOSE	Use to determine whether the drive is local or redirected to a remote.
CALL	AX = 4409H — Get whether drive is local or remote BL — Drive (0=Default,1=A,2=B,...)
RETURN	Carry set: AX = 1 — Invalid function AX = 15 — Invalid drive Carry not set: DX — Device attribute bits
MACRO	<pre>ioctl_rblock macro drive mov ax,4409H mov bl,drive int 21H endm</pre>
COMMENTS	<p>This function checks whether a drive is local or redirected to a remote. The BL register must contain the drive number.</p> <p>If the block device is local, DX returns the attribute word from the device header. If the block device is remote, only bit 12 is set (=1). The other bits are "0".</p> <p>An application program should not test bit 12. Applications should make no distinction between local and remote files or devices.</p>

EXAMPLE

The following program checks whether drive B is local or remote, and displays the appropriate message:

Note: This example must have function 40H loaded to work properly.

```

stdout equ 1
;
message db "Drive B: is "
loc db "local."
rem db "remote."
crlf db 0DH,0AH
;
begin: write_handle stdout,message,0CH ;display message
      jc write_error ;not shown
      ioctl_rblock 2 ;THIS FUNCTION
      jc ioctl_error ;not shown
      test dx,1000h ;bit 12 set?
      jnz not_loc ;yes, it's remote
      write_handle stdout,loc,6 ;Function 40H
      jc write_error ;not shown
      jmp done
not_loc: write_handle stdout,rem,7 ;Function 40H
      jc write_error ;not shown
done: write_handle stdout,crlf,2 ;Function 40H
      jc write_error ;not shown
      end

```

440AH - IOCTL is Redirected Handle

PURPOSE Use to determine whether the handle is local or redirected to a remote.

CALL AX = 440AH — Get whether drive is local or remote
BX — Handle

RETURN Carry set:
AX = 1 — Invalid function
AX = 6 — Invalid handle

Carry not set:
DX — IOCTL bit field

MACRO

```
ioctl_rhandle macro handle
    mov     ax, 440AH
    mov     bx, handle
    int     21H
endm
```

COMMENTS This function checks whether a handle is local or redirected to a remote. The BX register must contain the file handle.

DX returns the IOCTL bit field. Bit 15 is set (=1) if the handle refers to a remote file or device. An application program should not test bit 15. Applications should make no distinction between local and remote files or devices.

EXAMPLE

The following program checks whether handle 5 refers to a local or remote file or device, then displays the appropriate message:

Note: This example must have function 40H loaded to work properly.

```

stdout    equ        1
;
message   db          "Handle 5 is "
loc       db          " local."
rem       db          " remote."
crlf      db          0DH,0AH
;
begin:    write_handle stdout,message,0CH    ;display message
          jc          write_error            ;not shown
          ioctl_rhandle 5                    ;THIS FUNCTION
          jc          ioctl_error            ;not shown
          test        dx,1000h                ;bit 12 set?
          jnz         not_loc                ;yes, it's remote
          write_handle stdout,loc,6          ;Function 40H
          jc          write_error            ;not shown
          jmp         done
not_loc:   write_handle stdout,rem,7          ;Function 40H
          jc          write_error            ;not shown
done:      write_handle stdout,crlf,2        ;Function 40H
          jc          write_error            ;not shown
          end

```

440BH - IOCTL Retry

PURPOSE	Use to set how many times MS-DOS should "retry" a disk operation that fails because of a file-sharing violation.
CALL	AX = 440BH — Set number of retries DX — Number of retries (count) CX — Wait time (pause)
RETURN	Carry set: AX = 1 — Invalid function Carry not set: No error
MACRO	<pre>ioctl_retry macro count,pause mov ax,440BH mov bx,count mov cx,pause int 21H endm</pre>
COMMENTS	MS-DOS retries a disk operation that fails because of a file-sharing violation three times unless this function is used to specify a different number. After the specified number of retries, MS-DOS issues interrupt 21H for the requesting process.

The effect of the delay parameter in the CX register is machine dependent. It specifies how many times MS-DOS should execute an empty loop. The actual time varies, depending on the processor and clock speed. You can determine the effect on your machine by using DEBUG to set the retries to "1" and timing several values in the CX register.

EXAMPLE

The following program sets the number of sharing retries to 10 and specifies a delay of 1000 between retries:

```
begin:  ioctl_retry    10,1000      ;THIS FUNCTION
        jc            error        ;not shown
        end
```

45H - Duplicate File Handle

PURPOSE	Use to duplicate a new handle for an opened file.
CALL	AH = 45H BX — File Handle
RETURN	Carry set: AX = 4 — Too many open files AX = 6 — Invalid handle Carry not set: AX — New file handle
MACRO	<pre>xdup macro handle mov bx,handle mov ah,45H int 21H endm</pre>
COMMENTS	Function 45H takes an already opened file handle and returns a new handle that refers to the same file at the same position.

Note: If the read/write pointer of either handle is moved the pointer for the other handle is also moved.

EXAMPLE

The following program defines standard output (handle 1) to a file named DIRFILE, invokes a second copy of COMMAND.COM to list the directory (which writes the directory to DIRFILE), then restores standard output to handle 1:

Note: This example must have functions 3CH, 3EH, 46H, 4AH, and 4BH loaded to work properly.

```

pgm_file      db      "command.com",0
cmd_line      db      9," /c dir /w",ODH
parm_blk      db      0EH dup (0)
path          db      "dirfile",0
dir_file      dw      ?                ;for handle
sav_stdout    dw      ?                ;for handle
;
begin: set_block      last_inst      ;Function 4AH
      jc      error_setblk      ;not shown
      create_handle path,0      ;Function 3CH
      jc      error_create      ;not shown
      mov     dif_file,ax      ;save handle
      xdup    1                ;THIS FUNCTION
      jc      error_xdup      ;not shown
      mov     sav_stdout,ax    ;save handle
      xdup2   dir_file,1      ;THIS FUNCTION
      jc      error_xdup2     ;not shown
      exec    pgm_file,cmd_line,parm,blk ;Function 4BH
      jc      error_exec      ;not shown
      xdup2   sav_stdout,1    ;Function 46H
      jc      error_xdup2     ;not shown
      close_handle sav_stdout ;Function 3EH
      jc      error_close     ;not shown
      close_handle dir_file   ;Function 3EH
      jc      error_close     ;not shown
      end

```

46H - Force Duplicate of a Handle

PURPOSE	Use to duplicate a new handle for an opened file with the new file handle located in the CX register.
CALL	AH = 46H BX — Existing file handle CX — New file handle
RETURN	Carry set: AX = 6 — Invalid handle AX = 4 — Too many open files Carry not set: No error
MACRO	<pre>xdup2 macro oldhandle,newhandle mov bx,oldhandle mov cx,newhandle mov ah,46H int 21H endm</pre>
COMMENTS	Function 46H takes an already opened file handle and returns a new handle that refers to the same file at the same position. If there was already a file open on handle CX, it is closed first.

Note: If the read/write pointer of either handle is moved the pointer for the other handle is also moved.

EXAMPLE

The following program defines standard output (handle 1) to a file named DIRFILE, invokes a second copy of COMMAND.COM to list the directory (which writes the directory to DIRFILE), then restores standard output to handle 1:

Note: This example must have functions 3CH, 3EH, 46H, 4AH, and 4BH loaded to work properly.

```

pgm_file      db      "command.com",0
cmd_line      db      9," /c dir /w",0DH
parm_blk      db      0EH dup (0)
path          db      "dirfile",0
dir_file      dw      ?           ;for handle
sav_stdout    dw      ?           ;for handle
;
begin: set_block      last_inst      ;Function 4AH
      jc              error_setblk    ;not shown
      create_handle path,0           ;Function 3CH
      jc              error_create    ;not shown
      mov             dif_file,ax     ;save handle
      xdup            1               ;Function 45H
      jc              error_xdup      ;not shown
      mov             sav_stdout,ax   ;save handle
      xdup2           dir_file,1      ;THIS FUNCTION
      jc              error_xdup2     ;not shown
      exec            pgm_file,cmd_line,parm,blk ;Function 4BH
      jc              error_exec      ;not shown
      xdup2           sav_stdout,1    ;THIS FUNCTION
      jc              error_xdup2     ;not shown
      close_handle    sav_stdout      ;Function 3EH
      jc              error_close     ;not shown
      close_handle    dir_file        ;Function 3EH
      jc              error_close     ;not shown
      end

```

47H - Get Current Directory Path

PURPOSE Use to determine the current directory path.

CALL AH = 47H
 DS:SI — Pointer to 64-byte memory area
 DL — Drive number

RETURN Carry set:
 AX = 15 — Invalid drive

 Carry not set:
 No error

MACRO `get_dir macro pointer,drive`
 `mov si,offset pointer`
 `mov dl,drive`
 `mov ah,47H`
 `int 21H`
 `endm`

COMMENTS The directory is root-relative and does not contain the drive specifier or leading path separator. The drive code passed in DL is 0=default, 1=A, 2=B, etc.

EXAMPLE

The following program displays the current directory on the disk in drive B:

Note: This example must have function 09H loaded to work properly.

```
disk      db      "b:\$"
buffer    db      40H dup (?)
;
begin:    get_dir  2,buffer      ;THIS FUNCTION
          jc      error_dir     ;not shown
          display  disk         ;Function 09H
          display_asciiz buffer ;end chapter
          end
```

48H - Allocate Memory

PURPOSE	Use to allocate free blocks of memory. The BX register contains the size.
CALL	AH = 48H BX — Size of memory to be allocated
RETURN	Carry set: AX = 7 — Area trashed AX = 8 — Not enough memory BX — Maximum size that could be allocated Carry not set: AX — Pointer to be allocated memory
MACRO	<pre>allocate_memory macro size mov bx, size mov ah, 48H int 21H endm</pre>
COMMENTS	Function 48H returns a pointer to a free block of memory that has the requested size in paragraphs.

Note: The “area trashed” means that the memory manager allocation marks have been destroyed.

EXAMPLE

The following program opens the file named TEXTFILE.ASC, calculates its size (Function 42H [Move a File Pointer]), allocates a block of memory the correct size, reads the file into the allocated memory, then frees the allocated memory:

Note: This example must have functions 40H, 42H, 49H, and 4AH loaded to work properly.

```

path      db      "textfile.asc",0
msg1      db      "File loaded into allocated memory block.",0DH,0AH
msg2      db      "Allocated memory now being freed.",0DH,0AH
handle    dw      ?
mem_seg    dw      ?
file_len   dw      ?
;
begin:    open_handle    path,0
          jc              error_open      ;not shown
          mov             handle,ax       ;save handle
          move_ptr        handle,0,0,2    ;Function 42H
          jc              error_move      ;not shown
          mov             file_len,ax     ;save file length
          set_block       last_inst       ;Function 4AH
          jc              error_set_block ;not shown
          allocate_memory file_len        ;THIS FUNCTION
          jc              error_allocate ;not shown
          mov             mem_seg,ax      ;save address of new memory
          move_ptr        handle,0,0,0    ;Function 42H
          jc              error_move      ;not shown
          push            ds              ;save DS
          mov             ax,mem_seg       ;get segment of new memory
          mov             ds,ax           ;set DS at new memory
          read_handle     cs:handle,0,cs:file_len ;read file into
                                          ; new memory
          pop             ds              ;restore DS
          jc              error_read      ;not shown
          (*** ENTER CODE TO PROCESS FILE HERE ***)
          write_handle     stdout,msg1,42H ;Function 40H
          jc              write_error     ;not shown
          free_memory      mem_seg        ;Function 49H
          jc              error_freemem    ;not shown
          write_handle     stdout,msg2,49H ;Function 40H
          jc              write_error     ;not shown
          end

```

49H - Free Allocated Memory

PURPOSE	Use to free blocks of memory that were allocated by function 48H.
CALL	AH = 49H ES — Segment address of memory area to be freed
RETURN	Carry set: AX = 7 — Area trashed AX = 9 — Invalid block Carry not set: No error
MACRO	<pre>free_memory macro address mov es,address mov ah,49H int 21H endm</pre>
COMMENTS	Function 49H returns a piece of previously allocated memory to the system pool.

EXAMPLE

The following program opens the file named TEXTFILE.ASC, calculates its size (Function 42H [Move a File Pointer]), allocates a block of memory the correct size, reads the file into the allocated memory, then frees the allocated memory:

Note: This example must have functions 40H, 42H, 48H, and 4AH loaded to work properly.

```

path      db      "textfile.asc",0
msg1      db      "File loaded into allocated memory block.",0DH,0AH
msg2      db      "Allocated memory now being freed.",0DH,0AH
handle    dw      ?
mem_seg    dw      ?
file_len   dw      ?
;
begin:  open_handle    path,0
        jc             error_open          ;not shown
        mov            handle,ax           ;save handle
        move_ptr       handle,0,0,2        ;Function 42H
        jc             error_move          ;not shown
        mov            file_len,ax         ;save file length
        set_block      last_inst           ;Function 4AH
        jc             error_set_block     ;not shown
        allocate_memory file_len           ;Function 48H
        jc             error_allocate      ;not shown
        mov            mem_seg,ax          ;save address of new memory
        move_ptr       handle,0,0,0        ;Function 42H
        jc             error_move          ;not shown
        push          ds                   ;save DS
        mov            ax,mem_seg          ;get segment of new memory
        mov            ds,ax               ;set DS at new memory
        read_handle    cs:handle,0,cs:file_len ;read file into
        ; new memory
        pop            ds                   ;restore DS
        jc             error_read          ;not shown
        (** ENTER CODE TO PROCESS FILE HERE **)
        write_handle   stdout,msg1,42H    ;Function 40H
        jc             write_error         ;not shown
        free_memory     mem_seg            ;THIS FUNCTION
        jc             error_freemem       ;not shown
        write_handle    stdout,msg2,49H    ;Function 40H
        jc             write_error         ;not shown
        end

```

4AH - Set Block

PURPOSE Use to modify the size of the allocated blocks of memory.

CALL AH = 4AH
 ES — Segment address of memory area
 BX — Requested memory area size

RETURN Carry set:
 AX = 7 — Area trashed
 AX = 8 — Not enough memory
 AX = 9 — Invalid block

 BX — Maximum size possible

 Carry not set:
 No error

MACRO `set_block macro address,size`
 `mov es,address`
 `mov bx,size`
 `mov ah,4AH`
 `int 21H`
 `endm`

COMMENTS Function 4AH will attempt to grow/shrink an allocated block of memory.

EXAMPLE

The following program invokes a second copy of COMMAND.COM and executes a directory command:

Note: This example must have function 4BH loaded to work properly.

```
pgm_file      db      "command.com",0
cmd_line      db      9," /c dir /w",0DH
parm_blk      db      0EH dup (?)
reg_save      db      0AH dup (?)
;
begin: set_block last_inst      ;THIS FUNCTION
      exec      pgm_file,cmd_line,parm_blk,0 ;Function 4BH
      end
```

4B00H - Load and Execute a Program

PURPOSE Use to load another program into memory and begin execution.

CALL AX = 4B00H
 ES:BX — Pointer to parameter block
 DS:DX — Pointer to pathname

RETURN Carry set:
 AX = 1 — Invalid function
 AX = 2 — File not found
 AX = 8 — Not enough memory
 AX = 10 — Bad environment
 AX = 11 — Bad format

 Carry not set:
 No error

MACRO `exec macro pointer,address`
 `mov dx,pointer`
 `mov bx,address`
 `mov ax,4B03H`
 `int 21H`
 `endm`

COMMENTS This function allows a program to load another program into memory and (default) begin execution to it.

 DS:DX points to the ASCIIZ name of the file to be loaded. ES:BX pointer to a parameter block for the load.

All open files of a process are duplicated in the child process after an EXEC. This is extremely powerful; the parent process has control over the meanings of stdin, stdout, stderr, stdaux and stdprn. The parent could, for example, write a series of records to a file, open the file as standard input, open a listing file as standard output and then EXEC a sort program that takes its input from stdin and writes to stdout.

Also inherited (or passed from the parent) is an "environment." This is a block of text strings (less than 32K bytes total) that convey various configuration parameters.

The format of the environment is as follows:

Offset	Contents
0	String 1
1	String 2
	.
	.
	.
	String n
	00

To find the address of a particular string in the environment, subtract 1 from the offset shown and add this to the **environment segment** in the parameter block.

Typically the environment strings have the form:

PARAMETER = VALUE

For example, COMMAND.COM might pass its execution search path as:

PATH=A:IN;B:\GWBASIC\LIB

A zero value of the environment address causes the child process to inherit the parent's environment unchanged.

The inputs to AL are function numbers, for which there are returns. The function number values and functions are as follows:

Parameter Block Contents

The parameter block can be found starting at the BX register offset, in this format:

Offset	Contents
0	Environment Segment
2	Pointer to Command Line (80H)
6	Pointer to Default FCB (Passed at 5CH)
10	Pointer to Default FCB (Passed at 6CH)

To find the starting address of an item in the parameter block, add the offset shown to the value in the BX register. See "FILE CONTROL BLOCKS" in Chapter 5 for more information about the contents of an FCB.

EXAMPLE

The following program invokes a second copy of COMMAND.COM and executes a directory command:

Note: This example must have function 4AH loaded to work properly.

```
pgm_file      db      "command.com",0
cmd_line      db      9," /c dir /w",0DH
parm_blk      db      0EH dup (?)
reg_save      db      0AH dup (?)
;
begin: set_block last_inst      ;Function 4AH
      exec      pgm_file,cmd_line,parm_blk,0 ;THIS FUNCTION
      end
```

4B03H - Load Overlay

PURPOSE	Use to load a program segment (overlay).
CALL	AX — 4B03H ES:BX — Pointer to parameter block (block) DS:DX — Pointer to pathname (path)
RETURN	Carry set: AX = 1 — Invalid function AX = 2 — File not found AX = 8 — Not enough memory AX = 10 — Bad environment Carry not set: No error
MACRO	<pre>exec_ovl macro block,path,seg_adrs mov bx,block mov dx,path mov block,seg_adrs mov block[02H],seg_adrs mov ax,4B03H int 21H endm</pre>
COMMENTS	MS-DOS assumes that the invoking program is loading into it's own address space, so no free memory is required. A program segment prefix is not created. The DX register must contain the offset of an ASCIIZ string that specifies the drive and pathname of the program file.

**Parameter
Block
Contents**

The parameter block can be found starting at the BX register offset, in this format:

Offset	Contents
0	Loading Address of File
2	Relocation Factor for Image

To find the starting address of an item in the parameter block, add the offset shown to the value in the BX register. See "FILE CONTROL BLOCKS" in Chapter 5 for more information about the contents of an FCB.

Loading Address of File

This word contains the segment address of where the file is to be loaded.

Relocation Factor for Image

This word contains the relocation factor to be applied to the image.

EXAMPLE

The following program opens a file named TEXTFILE.ASC, redirects standard input to that file, loads MORE.COM as an overlay, and calls MORE.COM. MORE.COM reads TEXTFILE.ASC as standard input:

Note: This example must have functions 3DH, 3EH, 45H, 48H, 49H, and 4AH loaded to work properly.

```
stdin    equ        0
;
file      db        " TEXTFILE.ASC" ,0
cmd_file  db        "\more.com",0
parm_blk  dw        4 dup (?)
handle    dw        ?
new_mem   dw        ?
;
begin:    set_block   last inst      ;Function 4AH
          jc          setblock_error ;not shown
          allocate_memory 2000      ;Function 48H
          jc          allocate_error ;not shown
          mov         new_mem,ax     ;save segment of memory
          open_handle  file,0        ;Function 3DH
          jc          open_error     ;not shown
          mov         handle,ax      ;save handle
          xdup2        handle,stdin  ;Function 46H
          jc          dup2_error     ;not shown
          close_handle handle        ;Function 3EH
          jc          close_error    ;not shown
          mov         ax,new_mem     ;adrs of new memory
          exec_ovl     cmd_file,parm_blk,ax ;THIS FUNCTION
          jc          exec_error     ;not shown
          mov         ax,new_mem     ;point to overlay
          sub         ax,10H         ;no PSP for overlay
          mov         ds,ax          ;DS for overlay
          call        cs:overlay     ;call overlay
          push        cs             ;restore DS to original
          pop         ds             ; segment
          free_memory  new_mem       ;Function 49H
          jc          free_error     ;not shown
          end
```

4CH - End Process

PURPOSE Use to terminate a current process.

CALL AH = 4CH
 AL — Return code

RETURN None

MACRO `end_process macro code`
 `mov al,code`
 `mov ah,4CH`
 `int 21H`
 `endm`

COMMENTS Function 4CH terminates the current process and transfers control to the invoking process. In addition, a return code may be sent. All files open at the time are closed.

This method is preferred over all others (Interrupt 20H, JMP 0) and has the advantage that CS:0 does not have to point to the Program Header Prefix.

EXAMPLE

The following program displays a message and returns to MS-DOS with a return code of 8. It uses only the opening portion of the sample program skeleton shown at the beginning of this chapter.

Note: This example must have function 09H loaded to work properly.

```
message db      "Displayed by FUNC_4CH example",0DH,0AH," $"
;
begin: display    message    ;Function 09H
      end_process 8          ;THIS FUNCTION
code   ends
      end          code
```


4DH - Get Return Code of Child Process

PURPOSE Use to determine the exit code specified by a child process.

CALL AH = 4DH

RETURN AX — Exit code

MACRO `ret_code macro`
 `mov ah,4DH`
 `int 21H`
 `endm`

COMMENTS Function 4DH returns the Exit code specified by a child process. It returns this Exit code only once. The low byte of this code is that sent by the Exit routine. The high byte (AH register) is one of the following:

Code	Meaning
------	---------

0	Terminate/abort
1	CTRL-C
2	Hard error
3	Terminate and stay resident (31H)

EXAMPLE None

4EH - Find First File

PURPOSE	Use to find all files that match the specified pathname including wild-cards.
CALL	AH = 4EH CX — Search attributes DS:DX — Pointer to pathname
RETURN	Carry set: AX = 2 — File not found AX = 18 — No more files Carry not set: No error
MACRO	<pre>find_first_file macro attribute,pointer mov cx,attribute mov dx,pointer mov ah,4EH int 21H endm</pre>
COMMENTS	Function 4EH takes a pathname with wild-card characters in the last component (passed in DS:DX), a set of attributes (passed in CX) and attempts to find all files that match the pathname and have a subset of the required attributes. A datablock at the current DMA is written that contains information in the following form:

```
find_buf_reserved DB 21 DUP (?); Reserved*
find_buf_attr     DB ? ;attribute found
find_buf_time     DW ? ;time
find_buf_date     DW ? ;date
find_buf_size_l   DW ? ;low(size)
find_buf_size_h   DW ? ;high(size)
find_buf_pname    DB 13 DUP (?) ;packed name
find_buf ENDS
```

*Reserved for MS-DOS use on subsequent
find_nexts

To obtain the subsequent matches of the
pathname, see the description of Function 4FH.

EXAMPLE

The following program displays a message that
specifies whether a file named REPORT.ASM
exists in the current directory on the disk in
drive B:

Note: This example must have
functions 09H and 1AH loaded to work
properly.

```
yes      db      " FILE EXISTS." ,ODH,0AH," $"
no       db      " FILE DOES NOT EXIST." ,ODH,0AH," $"
path     db      "b:report.asm",0
buffer   db      26H dup (?)
;
begin:   set_dta   buffer                ;Function 1AH
         find_first_file path,0          ;THIS FUNCTION
         jc       error_findfirst       ;not shown
         cmp      al,12H                 ;file found?
         je       not_there              ;no
         display  yes                     ;Function 09H
         jmp      return                 ;all done
not_there: display no                     ;Function 09H
         end
```

4FH - Find Next File

PURPOSE	Use to advance to the next matching file started with function 4EH.
CALL	AH = 4FH
RETURN	Carry set: AX = 18 — No more files Carry not set: No error
MACRO	<pre>find_next_file macro mov ah,4FH int 21H endm</pre>
COMMENTS	The current DMA address must point at a block returned by Function 4EH (see Function 4EH).

EXAMPLE

The following program displays the number of files in the current directory on the disk in drive B:

Note: This example must have functions 09H, 1AH, and 4EH loaded to work properly.

```

message      db          " No files",0DH,0AH," $"
files        dw          ?
path         db          "b:*.\"",0
buffer       db          26H dup (?)
;
begin:       set_dta     buffer           ;Function 1AH
             find_first_file path,0      ;Function 4EH
             jc          error_findfirst ;not shown
             cmp         al,12H          ;directory empty?
             je          all_done        ;yes, go home
             inc         files           ;no, bump file counter
search_dir:  find_next_file              ;THIS FUNCTION
             jc          error_findnext  ;not shown
             cmp         al,12H          ;any more entries?
             je          done            ;no, go home
             inc         files           ;yes, bump file counter
             jmp         search_dir      ;and check again
done:        convert     files,10,message ;end of chapter
all_done     display     message         ;Function 09H
             end

```

54H - Get Verify State

PURPOSE	Use to determine the current status of the verify flag. The status is located in the AL register.
CALL	AH = 54H
RETURN	AL — Current verify flag value
MACRO	<pre> get_verify macro mov ah,54H int 21H endm </pre>
COMMENTS	The current value of the verify flag is returned in AL.
EXAMPLE	The following program displays the verify status:

Note: This example must have functions 09H loaded to work properly.

```

message db      "Verify ","$"
on       db      "on.",0DH,0AH,"$"
off      db      "off.",0DH,0AH,"$"
;
begin:  display   message      ;Function 09H
        get_verify      ;THIS FUNCTION
        cmp         al,0      ;is flag off?
        jg          ver_on    ;no, it's on
        display      off      ;Function 09H
        jmp         return    ;go home
ver_on: display      on        ;Function 09H
        end

```

56H - Change Directory Entry

PURPOSE Use to move a file to another directory (path).

CALL AH = 56H
 ES:DI — Pointer to new pathname
 DS:DX — Pointer to pathname of existing file

RETURN Carry set:
 AX = 2 — File not found
 AX = 5 — Access denied
 AX = 17 — Not same device

 Carry not set:
 No error

MACRO `rename_file macro oldpointer,newpointer`
 `mov dx,offset oldpointer`
 `mov di,offset newpointer`
 `mov ah,56H`
 `int 21H`
 `endm`

COMMENTS Function 56H attempts to rename a file into another path. The paths must be on the same device.

EXAMPLE The following program prompt for the name of a file and a new name, then renames the file:

Note: This example must have functions 09H and 0AH loaded to work properly.

```
prompt1 db "Filename: $"
prompt2 db "New name: $"
old_path db 0FH,?,0FH dup (?)
new_path db 0FH,?,0FH dup (?)
crlf db 0DH,0AH," $"
;
begin:  display prompt1           ;Function 09H
        get_string 0FH,old_path   ;Function 0AH
        xor bx,bx                 ;to use BL as index
        mov bl,old_path[1]        ;get string length
        mov old_path[bx+2],0      ;make an ASCIZ string
        display crlf             ;Function 09H
        display prompt2          ;Function 09H
        get_string 0FH,new_path   ;Function 0AH
        xor bx,bx                 ;to use BL as index
        mov bl,new_path[1]        ;get string length
        mov new_path[bx+2],0      ;make an ASCIZ string
        display crlf             ;Function 09H
        rename_file old_path[2],new_path[2] ;THIS FUNCTION
        jc error_rename          ;not shown
end
```


57H - Get/Set Date/Time of File

PURPOSE Use to determine or set the last-write time for a handle.

CALL AH = 57H

AL = 0 — Get date and time

AL = 1 — Set date and time

BX — File handle

CX — Time to be set (When AL=1 only)

DX — Date to be set (When AL=1 only)

RETURN Carry set:

AX = 1 — Invalid function

AX = 6 — Invalid handle

Carry not set:

No error

CX — Set when AL=0 only

DX — Set when AL=0 only

MACRO

get_set_date_time macro function,handle
[,time,date]

```
mov    bx,handle
mov    cx,time
mov    dx,date
mov    al,function
mov    ah,57H
int    21H
endm
```

COMMENTS	Function 57H returns or sets the last-write time for a handle. These times are not recorded until the file is closed. The inputs to AL are function numbers, for which there are returns. The function number values and functions are as follows:
Function 0	This function gets the time (CX) and date (DX) of the handle.
Function 1	This function sets the time (CX) and date (DX) of the handle.

EXAMPLE

The following program gets the date of the file named REPORT.ASM in the current directory on the disk in drive B, increments the day, increments the month and/or year (if necessary), and sets the new date of the file:

Note: This example must have functions 3DH and 3EH loaded to work properly.

```

month      db      31,28,31,30,31,30,31,31,30,31,30,31
path       db      "b:report.asm",0
handle     dw      ?
time       db      2 dup (?)
date       db      2 dup (?)
;
begin:     open_handle path,0           ;Function 3DH
           mov     handle,ax           ;save handle
           get_set_date_time handle,0,time,date;THIS FUNCTION
           jc      error_time         ;not shown
           mov     word ptr time,cx    ;save time
           mov     word ptr date,dx    ;save date
           convert_date date[-24]     ;end of chapter
           inc     dh                  ;increment day
           xor     bx,bx               ;to use BL as index
           mov     bl,dh              ;get month
           cmp     dh,month[bx-1]     ;past last day?
           jle     month_ok           ;no, go home
           mov     dh,1               ;yes, set day to 1
           inc     dl                  ;increment month
           cmp     dl,12               ;is it past December?
           jle     month_ok           ;no, go home
           mov     dl,1               ;yes, set month to 1
           inc     cx                  ;increment year
month_ok:   pack_date date            ;end of chapter
           get_set_date_time handle,1,time,date ;THIS FUNCTION
           jc      error_time         ;not shown
           close_handle handle       ;Function 3EH
           jc      error_close        ;not shown
           end

```

58H - Get/Set Allocation Strategy

PURPOSE Use to get/set the memory allocation strategy.

CALL AH = 58H

AL = 0 — Get allocation strategy

AL = 1 — Set allocation strategy

BX = 0 — First fit

BX = 1 — Best fit

BX = 2 — Last fit

RETURN Carry set:
 AX = 1 — Invalid function

Carry not set:
 AX = 0 — First fit (AL=1)
 AX = 1 — Best fit (AL=1)
 AX = 2 — Last fit (AL=1)

MACRO alloc_strat macro function, strategy
 mov al, function
 mov bx, strategy
 mov ah, 58H
 int 21H
 endm

COMMENTS

Function 58H gets or sets the strategy used by MS-DOS to allocate memory, when requested by a process. If the AL register contains "0", the strategy is returned in the AX register. If the AL register contains "1", the BX register must indicate the strategy. There are three possible strategies:

- **First fit (0)** — MS-DOS starts searching at the lowest available block of memory and allocates the first block it finds (the allocated memory is the lowest available block). This is the default strategy.
- **Best fit (1)** — MS-DOS searches each available block and allocates the smallest available block that satisfies the request.
- **Last fit (2)** — MS-DOS starts searching at the highest available block and allocates the first block it finds (the allocated memory is the highest available block).

Basically, this function can be used to control how MS-DOS uses its memory resources.

EXAMPLE

The following program displays the memory allocation strategy in effect, then forces subsequent memory allocations to the top of memory by setting the strategy to last fit (code 2):

```
get      equ      0
set      equ      1
stdout   equ      1
last_fit equ      2
;
first    db        "First fit      ",0DH,0AH
best     db        "Best fit       ",0DH,0AH
last     db        "Last fit       ",0DH,0AH
;
begin:   alloc_strat get          ;THIS FUNCTION
        jc         alloc_error    ;not shown
        mv         cl,4           ;multiply code by 16
        shl        ax,cl          ;to calculate offset
        mov        dx,offset first;point to first msg
        add        dx,ax          ;add to base address
        mov        bx,stdout      ;handle for write
        mov        cs,10H         ;write 16 bytes
        mov        ah,40h         ;write handle
        int        21H           ;system call
;
        jc         write_error    ;not shown
        alloc_strat set,last-fit   ;THIS FUNCTION
;
        jc         alloc_error    ;not shown
        end
```

59H - Get Extended Error

PURPOSE Use to get the extended error code for the previously processed function.

CALL AH = 59H
 BX = 0

RETURN AX — Extended error code
 BH — Error class
 BL — Suggested action
 CH — Locus

Note: The contents of registers CL, DX, SI, DI, BP, DS, and ES are destroyed.

MACRO `get_error macro`
 `mov al,function`
 `mov bx,00H`
 `mov ah,59H`
 `int 21H`
 `endm`

COMMENTS Function 59H gets an extended error code for the previous function. These new codes are mapped to a simpler set of error codes based on MS-DOS version 2.0. Therefore, existing programs will operate correctly.

An interrupt 24H handles can use this function to get detailed information about the error that caused the interrupt to be issued.

The BX register contains a version indicator which specifies the level of error handling the application was written for. The current level is "0".

The extended error code is composed of four separate codes, in the AX, BH, BL, and CH registers, that provide as much detail as possible about the error. Also, suggesting how the issuing program should respond.

BH — Error Class

The BH register contains the error class code as follows:

Code	Description
1	Out of a resource, such as storage or channels
2	Not an error; a temporary situation (i.e., a locked region in a file) that can be expected to end
3	Authorization problem
4	An internal error in system software
5	Hardware failure
6	System software failure not the fault of the active process; could be caused by missing or incorrect configuration files
7	Application program error
8	File or item not found
9	File or item of invalid format, type, or otherwise invalid or unsuitable
10	File or item interlocked
11	Wrong disk in drive, bad spot on disk, or other problem with storage media
12	Other error

**BL —
Suggested
Action**

The BL register contains the suggested action code as follows:

Code	Description
1	Retry, then prompt user
2	Retry, after a pause
3	If user entered data such as a drive or filename, prompt for it again
4	Terminate with cleanup
5	Terminate immediately. The system is so unhealthy that the program should exit as soon as possible, without taking the time to close files and update indexes
6	Error is informational
7	Prompt user to perform some action, such as changing disks, then retry the operation

CH — Locus

The CH register contains additional information to help locate the area involved in the failure. This code is particularly useful for hardware failures (BH=5).

Code	Description
1	Unknown
2	Related to random access block devices, such as a disk drive
3	Related to network
4	Related to serial access character devices, such as a printer
5	Related to random access memory

Your programs should handle errors by noting the occurrence of an error, then issuing the function to get the extended error code. If the program does not recognize the extended error code, it should respond to the original error code.

This function is available during interrupt 24H and may be used to return network-related errors.

EXAMPLE None

5AH - Create Temporary File

PURPOSE	Use to create a temporary unique file which is associated to the path specified by the DX register.
CALL	AH = 5AH CX — Attribute DS:DX — Pointer to pathname followed by a byte of "0", and 13 bytes of memory
RETURN	Carry set: AX = 3 — Path not found AX = 5 — Access denied Carry not set: AX — Handle
MACRO	<pre>create_temp macro attribute,path mov cx,attribute mov dx,offset path mov ah,5AH int 21H endm</pre>
COMMENTS	The DX register must contain the offset of an ASCIIZ string that specifies a pathname and 13 bytes of memory (for filename). The CX register must contain the attribute to be assigned to the file. The attribute format is in Chapter 5, Memory and Disk Allocation , under Disk Directory .

MS-DOS creates a unique filename and appends it to path specified by DS:DX, creates the file and opens it in compatibility mode, then returns the file handle in AX. A program that needs a temporary file should use this function to avoid name conflicts.

MS-DOS does **not** delete a file created by this function after the creating process exits. You should delete files no longer needed.

EXAMPLE

The following program creates a temporary file in the directory named \WP\DOCS, copies a file in the current directory named TEXTFILE.ASC into the temporary file, then closes both files:

Note: This example must have functions 02H, 3DH, 3EH, and 40H loaded to work properly.

```
stdout equ 1
;
file db "TEXTFILE.ASC",0
path db "\WP\DOCS",0
temp db 0DH dup (0)
open_msg db " opened.",0DH,0AH
crl_msg db " created.",0DH,0AH
rd_msg db " read into buffer.",0DH,0AH
wr_msg db " Buffer written to "
cl_msg db " Files closed.",0DH,0AH
crlf db 0DH,0AH
handle1 dw ?
handle2 dw ?
buffer db 200H dup (?)
;
begin: open_handle file,0 ;Function 3DH
      jc open_error ;not shown
      mov handle1,ax ;save handle
      write_handle stdout,file,0CH ;Function 40H
      jc write_error ;not shown
```

```

write_handle stdout,open_msg,0AH ;Function 40H
jc          write_error          ;not shown
create_temp path,0               ;THIS FUNCTION
jc          create_error         ;not shown
mov         handle2,ax           ;save handle
write_handle stdout,path,8       ;Function 40H
jc          write_error          ;not shown
display_char "\
write_handle stdout,temp,0CH     ;Function 40H
jc          write_error          ;not shown
write_handle stdout,crl_msg,0BH  ;Function 40H
jc          write_error          ;not shown
read_handle handle1,buffer,200H ;Function 3FH
jc          read_error           ;not shown
write_handle stdout,file,0CH     ;Function 40H
jc          write_error          ;not shown
write_handle stdout,rd_msg,14H   ;Function 40H
jc          write_error          ;not shown
write_handle handle2,buffer,200H ;Function 40H
jc          write_error          ;not shown
write_handle stdout,wr_msg,12H   ;Function 40H
jc          write_error          ;not shown
write_handle stdout,temp,0CH     ;Function 40H
jc          write_error          ;not shown
write_handle stdout,crlf,2       ;Function 40H
jc          write_error          ;not shown
close_handle handle1            ;Function 3EH
jc          write_error          ;not shown
close_handle handle2            ;Function 3EH
jc          write_error          ;not shown
write_handle stdout,cl_msg,0FH   ;Function 40H
jc          write_error          ;not shown
end

```

5BH - Create New File

PURPOSE Use to add a file to the directory specified by the ASCIIZ string indicated by the DX register. This function fails if the file already exists.

CALL AH = 5BH
CX — Attribute
DS:DX — Pointer to pathname

RETURN Carry set:
AX = 3 — Path not found
AX = 4 — Too many opened files
AX = 5 — Access denied
AX = 80 — File already exists

Carry not set:
AX — Handle

MACRO

```
create_new macro attribute,path
    mov     cx,attribute
    mov     dx,path
    mov     ah,5AH
    int     21H
endm
```

COMMENTS The DX register must contain the offset of an ASCIIZ string that specifies a pathname. The CX register must contain the attribute to be assigned to the file. The attribute format is in Chapter 5, **Memory and Disk Allocation**, under **Disk Directory**.

If there is no existing file with the same filename, MS-DOS creates the file and opens it in compatibility mode, then returns the file handle in AX.

Unlike Function 3CH (Create a file), this function fails if the file already exists, rather than truncating it to a length of zero. The existence of a file is used as a semaphore in a multi-tasking system; you can use this function as a test-and-set semaphore.

EXAMPLE

The following program attempts to create a new file named REPORT.ASM in the current directory. if the file already exists, the program displays an error message and returns to MS-DOS. if the file does not exist and there are no other errors, the program saves the handle and continues processing:

Note: This example must have function 09H loaded to work properly.

```

err_msg  db    " FILE ALREADY EXISTS" ,ODH,0AH," $"
path     db    "REPORT.ASM",0
handle   dw    ?
;
begin:   create_new path,0           ;THIS FUNCTION
jnc      jnc      continue          ;further processing
        cmp      ax,50H             ;file already exist?
        jne      error              ;not shown
        display   err_msg           ;Function 09H
        jmp      return             ;return to MS-DOS
continue: mov     handle,ax          ;save handle
;
;      ( *** FURTHER PROCESSING HERE *** )

```

5C00H - Lock

PURPOSE Use to deny (lock) all access (read/write) by any other process to a specified region of a file.

CALL AX = 5C00H
BX — Handle
CX:DX — Pointer to locked region (pointer)
SI:DI — Size of locked region (size)

RETURN Carry set:
AX = 1 — Invalid function
AX = 6 — Invalid handle
AX = 22 — Lock violation

Carry not set:
No error

MACRO

```
lock macro handle,lpointer,hpointer,lsize,hsize
    mov     bx,handle
    mov     cx,hpointer
    mov     dx,lpointer
    mov     si,hsize
    mov     di,lsize
    mov     ax,5C00H
    int     21H
endm
```

COMMENTS The BX register contains the handle of the file that contains the locked region. Registers CX and DX contain the offset in the file of the beginning of the region. Registers SI and DI contain the size of the region.

If another process attempts to use a “locked” region, MS-DOS retries up to three times. If the retries fail, MS-DOS issues interrupt 24H

for the requesting process. You can change the number of retries with Function 440BH (IOCTL Retry).

The locked region can be anywhere in the file. Locking beyond the end of a file is not an error. Regions should only be locked for brief periods; locking for more than 10 seconds should be considered an error.

Functions 45H (Duplicate a File Handle) and 46H (Force a Duplicate of a File Handle) duplicate access to any locked region. Passing an open file to a child process with Function 4BH (Load and Execute a Program) does **not** duplicate access to locked regions.

If a program closes a file or terminates with an open file that contains a locked region, the result is undefined. Programs that might be terminated by Interrupt 23H (Control-C) or 24H (a fatal error) should trap these interrupts and unlock any locked regions before exiting.

Programs should not rely on being denied access to a locked region. a program can determine the status of a region (locked/unlocked) by attempting to lock the region and examining the error code.

EXAMPLE

The following program opens a file named FINAL.RPT in Deny None mode and locks two portions of it: the first 128 bytes and bytes 1024 through 5119. After some processing, it unlocks the same portions and closes the file:

Note: This example must have functions 3DH, 40H, and 5C01H loaded to work properly.

```
stdout      equ      1
;
start1      dd      0
lgth1       dd      80H
start2      dd      1023
lgth2       dd      4096
file        db      "FINALRPT",0
op_msg      db      " opened.",0DH,0AH
l1_msg      db      "First 80H bytes locked.",0DH,0AH
l2_msg      db      "Bytes 1024-5119 locked.",0DH,0AH
u1_msg      db      "First 80H bytes unlocked.",0DH,0AH
u2_msg      db      "Bytes 1024-5119 unlocked.",0DH,0AH
cl_msg      db      " closed.: ,0DH,0AH
handle      dw      ?
;
begin:      open_handle file,01000010b ;Function 3DH
            jc      open_error          ;not shown
            write_handle stdout,file,8 ;Function 40H
            jc      write_error         ;not shown
            write_handle stdout,op_msg,0AH ;Function 40H
            jc      write_error         ;not shown
            mov     handle,ax           ;save handle
            lock    handle,start1,lgth1 ;THIS FUNCTION
            jc      lock_error          ;not shown
            write_handle stdout,OBH_msg,19H ;Function 40H
            jc      write_error         ;not shown
            lock    handle,start2,lgth2 ;THIS FUNCTION
            jc      lock_error          ;not shown
            write_handle stdout,0CH_msg,19H ;Function 40H
            jc      write_error         ;not shown
;
; (*** FURTHER PROCESSING HERE ***)
;
            unlock    handle,start1,lgth1 ;Function 5C01H
            jc      unlock_error          ;not shown
            write_handle stdout,u1_msg,1BH ;Function 40H
            jc      write_error         ;not shown
            unlock    handle,start2,lgth2 ;Function 5C01H
            jc      unlock_error          ;not shown
            write_handle stdout,u2_msg,1BH ;Function 40H
            jc      write_error         ;not shown
            close_handle handle          ;Function 3EH
            jc      close_error          ;not shown
            write_handle stdout,file,8 ;Function 40H
            jc      write_error         ;not shown
            write_handle stdout,cl_msg,0AH ;Function 40H
            jc      write_error         ;not show
end
```

5C01H - Unlock

PURPOSE Use to allow (unlock) access (read/write) to a locked region of a file. Use to deny (lock) all access (read/write) by any other process to a specified region of a file.

CALL AX = 5C01H
BX — Handle
CX:DX — Pointer to locked region (pointer)
SI:DI — Size of locked region (size)

RETURN Carry set:
AX = 1 — Invalid function
AX = 6 — Invalid handle
AX = 22 — Lock violation

Carry not set:
No error

MACRO unlock macro handle,lpointer,hpointer,
lsize,hsize
mov bx,handle
mov cx,hpointer
mov dx,lpointer
mov si,hsize
mov di,lsize
mov ax,5C01H
int 21H
endm

COMMENTS

The BX register contains the handle of the file that contains the region to be unlocked. Registers CX and DX contain the offset in the file of the beginning of the region. Registers SI and DI contain the size of the region. The offset and the size must be specified exactly as when the region was locked (Function 5C00H).

EXAMPLE

The following program opens a file named FINAL.RPT in Deny None mode and locks two portions of it: the first 128 bytes and bytes 1024 through 5119. After some processing, it unlocks the same portions and closes the file:

Note: This example must have functions 3DH, 40H, and 5C00H loaded to work properly.

```

stdout    equ    1
;
start1    dd     0
lgth1     dd     80H
start2    dd     1023
lgth2     dd     4096
file       db     "FINALRPT",0
op_msg    db     " opened.",0DH,0AH
l1_msg    db     "First 80H bytes locked.",0DH,0AH
l2_msg    db     "Bytes 1024-5119 locked.",0DH,0AH
ul_msg    db     "First 80H bytes unlocked.",0DH,0AH
u2_msg    db     "Bytes 1024-5119 unlocked.",0DH,0AH
cl_msg    db     " closed.: ",0DH,0AH
handle     dw     ?
;
begin:     open_handle file,01000010b    ;Function 3DH
           jc      open_error            ;not shown
           write_handle stdout,file,8    ;Function 40H
           jc      write_error           ;not shown
           write_handle stdout,op_msg,0AH ;Function 40H
           jc      write_error           ;not shown
           mov     handle,ax              ;save handle
           lock    handle,start1,lgth1   ;Function 5C00H
           jc      lock_error            ;not shown
           write_handle stdout,l1_msg,19H ;Function 40H
           jc      write_error           ;not shown
           lock    handle,start2,lgth2   ;Function 5C00H
           jc      lock_error            ;not shown
           write_handle stdout,l2_msg,19H ;Function 40H
           jc      write_error           ;not shown
;
; (*** FURTHER PROCESSING HERE ***)
;
           unlock   handle,start1,lgth1  ;THIS FUNCTION
           jc      unlock_error          ;not shown
           write_handle stdout,u1_msg,27 ;Function 40H
           jc      write_error           ;not shown
           unlock   handle,start2,lgth2  ;THIS FUNCTION
           jc      unlock_error          ;not shown
           write_handle stdout,u2_msg,27 ;Function 40H
           jc      write_error           ;not shown
           close_handle handle            ;Function 3EH
           jc      close_error           ;not shown
           write_handle stdout,file,8    ;Function 40H
           jc      write_error           ;not shown
           write_handle stdout,cl_msg,10 ;Function 40H
           jc      write_error           ;not shown
           end

```

5E00H - Get Machine Name

PURPOSE	Use to get the network name of the local computer.
CALL	AX = 5E00H BX — Handle DS:DX — Pointer to 16 byte buffer
RETURN	Carry set: AX = 1 — Invalid function Carry not set: CX — Identification number of local computer
MACRO	<pre>get_machine_name macro pointer mov dx,offset pointer mov ax,5E00H int 21H endm</pre>
COMMENTS	The DX register must contain the offset (pointer) of a 16 byte buffer for the local computer name. MS-DOS returns the computer name (16 byte ASCII string padded with blanks) to the buffer. The CX register contains the identification number of the local computer.

EXAMPLE

The following program displays the name of a Microsoft Networks station:

Note: This example must have function 40H loaded to work properly.

```
stdout equ    1
;
msg      db      "Netname: "
mac_name db      10H dup (?),ODH,0AH
;
begin:   get_machine_name mac_name      ;THIS FUNCTION
        jc              name_error    ;not shown
        write_handle     stdout,msg,1BH ;Function 40H
        jc              write_error   ;not shown
        end
```

5E02H - Printer Setup

PURPOSE	Use to define a string of control characters which MS-DOS will send to a network printer ahead of each file sent.
CALL	AX = 5E02H BX — Assign list index CX — Size of setup string DS:SI — Pointer to string
RETURN	Carry set: AX = 1 — Invalid function Carry not set: No error
MACRO	<pre>printer_setup macro index,size,pointer mov bx,index mov cx,size mov si,pointer mov ax,5E02H int 21H endm</pre>
COMMENTS	The BX register must contain the index into the assign list that identifies the printer (entry "0" is the first entry). The CX register must contain the size of the string. SI must contain the offset (to DS) of the string itself. Microsoft Networks must be running.

The setup string is added to the beginning of each file sent to the printer specified by the assign list index in BX. You can use this function to provide a shared printer with your printer configuration. Use Function 5F02H (Get Assign List Entry) to determine which entry in the assign list refers to the printer.

EXAMPLE

The following program defines a printer setup string that consists of the control character to print expanded type on AT&T 473 compatible printers. the printer cancels this mode at the first carriage return, so the effect is to print the first line of each file sent to the network printer to look like a large title. The setup string is one character. This program assumes that the printer is entry number 3 (fourth entry) in the assign list. Use Function 5F02H (Get Assign List Entry) to determine this value:

```
setup  db      0EH
;
begin: printer_setup 3,1,setup      ;THIS FUNCTION
      jc          error      ;not shown
      end
```

5F02H - Get Assign List Entry

PURPOSE Use to get the specified entry from the network list of assignments.

CALL AX = 5E02H
 DS:SI — Pointer to local name
 ES:DI — Pointer to remote name

RETURN Carry set:
 AX = 1 — Invalid function
 AX = 18 — No more files

Carry not set:
 BL = 3 — Printer
 BL = 4 — Drive

CX — Stored user value

MACRO

```

get_list macro index,hlocal,llocal,hremote,
           lremote
    mov     bx,index
    mov     ds,offset hlocal
    mov     si,offset llocal
    mov     es,offset hremote
    mov     di,offset lremote
    mov     ax,5F02H
    int     21H
endm

```

COMMENTS The SI register must contain the offset (to DS) of a 16 byte buffer for the local name. The DI register must contain the offset (to ES) of a 128 byte buffer for the remote name. Microsoft Networks must be running.

You can use this function to retrieve any entry, or make a copy of the complete list by stepping through the table. To detect the end of the assign list, check for error code 18 (No more files), just as when you step through a directory with Functions 4EH (Find Match File) and 4FH (Step Through a Directory Matching Files).

MS-DOS puts the local name in the buffer pointed to by SI and the remote name in the buffer pointed to by DI. The local name can be a null ASCII string. The BL register contains "3" if the local device is a printer, or "4" if the local device is a drive. CX returns the stored user value set with Function 5F03H (Make Assign List Entry). The contents of the assign list can change between functions.

EXAMPLE

The following program displays the assign list on a Microsoft Networks workstation. The assign list contains the local name, remote name, and device type (drive or printer) for each entry:

Note: This example must have function 40H loaded to work properly.

```
stdout      equ      1          ;code returned from
printer     equ      3          ;GetAssignListEntry for a printer
header      db        0DH,0AH,0DH,0AH,"Device Type"
            db        "Local name",9 dup (20h)
            db        "Remote name"
crlf        db        0DH,0AH,0DH,0AH
header_len  equ      $ - header

local_nm    db        13H dup (?)
remote_nm_len equ    $ - local_nm

remote_nm    db        80H dup (?)
remote_nm_len equ    $ - remote_nm

drive_msg    db        "Drive",8 dup (20h)
print_msg    db        "Printer", 6 dup (20h)
devices_msg_len equ    $ - print_msg

str_len     dw        ?
index       dw        ?
;
begin: write_handle stdout,header,header_len ;Function 40H
      jnc      set_index
      jmp      write_error
set_index: mov    index,0          ;assign list index
      end
```

5F03H - Make Assign List Entry

PURPOSE Use to redirect a printer or drive (source device) to a network directory (destination device).

CALL AX = 5E03H

 BL = 3 — Printer
 BL = 4 — Drive

 CX — User value
 DS:SI — Pointer to source
 ES:DI — Pointer to destination

RETURN Carry set:

 AX = 1 — Invalid function
 AX = 3 — Path not found
 AX = 5 — Access denied
 AX = 8 — Insufficient memory

Note: Other errors particular to the network may occur.

Carry not set:
 No error

MACRO redir macro device,value,source,destination

```

mov     bl,device
mov     cx,value
mov     si,offset source
mov     di,offset destination
mov     ax,5F03H
int     21H
endm
```

COMMENTS The SI register must contain the offset (to DS) of an ASCIIZ string that specifies either the name of the printer, a drive letter followed by a colon, or a null string. The DI register must contain the offset (to ES) of an ASCIIZ string that specifies the name of a network directory. Microsoft Networks must be running.

Source String If the BL register is "3", the source string must be PRN, LPT1, LPT2, or LPT3. All output for the named printer is buffered and sent to the remote spooler named in the destination string.

If the BL register is "4", the source string can be either a drive letter with a colon or a null string. If the source string contains a valid drive, all subsequent references to the drive letter are redirected to the network directory named in the destination string. If the source string is a null string, MS-DOS attempts to grant access to the network directory with the specified password.

Destination String The maximum length of the destination string is 128 bytes. The value in CX can be retrieved with Function 5F02H (Get Assign List Entry).

The destination string is as follows:

<machine_name><pathname><00H><password><00H

<machine_name> is the network name of the server that contains the network directory.

<pathname> is the alias of the network directory, **not** the directory path, to which the source device is to be redirected.

<00H> is the null byte.

<password> is the password for access to the network directory. If no password is specified, both null bytes must follow the <pathname>.

EXAMPLE

The following program redirects two drives and a printer from a workstation to a server named HAROLD. It assumes the machine name, directory names, and drive letters shown:

Local drive or printer	Netname on server	password
E:	WORD	none
F:	COMM	fred
PRN:	PRINTER	quick


```

printer equ 3
drive    equ 4
;
local_1  db  "e:",0
local_2  db  "f:",0
local_3  db  "prn",0
remote_1 db  "\old\
338remote_2 db  "\old\remote_3 db  "\old\rinter",0,"quick",0
;
begin:   redir  local_1,remote_1,drive,0    ;THIS FUNCTION
         jc     error                       ;not shown
         redir  local_2,remote_2,drive,0    ;THIS FUNCTION
         jc     error                       ;not shown
         redir  local_3,remote_3,printer,0  ;THIS FUNCTION
         jc     error                       ;not shown
end

```

5F04H - Cancel Assign List Entry

PURPOSE Use to cancel redirection of a printer or drive
 (source device) to a network directory
 (destination device).

CALL AX = 5E04H
 DS:SI — Pointer to destination

RETURN Carry set:
 AX = 1 — Invalid function
 AX = 15 — Redirection paused on server

Note: Other errors particular to
the network may occur.

Carry not set:
No error

MACRO `cancel_redir macro source`
 `mov si,offset source`
 `mov ax,5F04H`
 `int 21H`
 `endm`

COMMENTS

Function 5F04H cancels the redirection of a source device to a destination device that was made with Function 5F03H (Make Assign List Entry). Microsoft Networks must be running. The ASCII string (source) can contain one of three values:

- The letter of a redirected drive, followed by a colon. The redirection is canceled and the drive is restored to its physical meaning.
- The name of a redirected printer (PRN, LPT1, LPT2, or LPT3). The redirection is canceled and the printer name is restored to its physical meaning.
- A string starting with \\ (two backslashes). The connection between the local machine and the network directory is terminated.

EXAMPLE

The following program cancels the redirection of drives E and F and the printer (PRN) of a Microsoft Networks workstation. It assumes that these local devices were previously redirected:

```

local_1 db      "e:",0
local_2 db      "f:",0
local_3 db      "prn:",0
;
begin: cancel_redir local_1      ;THIS FUNCTION
      jc          error         ;not shown
begin: cancel_redir local_2      ;THIS FUNCTION
      jc          error         ;not shown
begin: cancel_redir local_3      ;THIS FUNCTION
      jc          error         ;not shown
end

```

62H - Get PSP

PURPOSE Use to get the segment address of the current (active) process (start of the Program Segment Prefix).

CALL AX = 62H

RETURN BX — Segment address of the PSP

MACRO `get_psp macro`
 `mov ax,62H`
 `int 21H`
 `endm`

COMMENTS The BX register contains the PSP.

EXAMPLE The following program displays the segment address of it's Program Segment Prefix (PSP). The address is returned in BX.

```
msg      db      " PSP segment address:  H",0DH,0AH," $"
;
begin:   get_psp                                ;THIS FUNCTION
convert  bx,10H,msg[15H]                       ;end of chapter
display  msg                                     ;Function 09H
end
```

General Macros

The following macros are provided to support some of the examples. They must be loaded for those examples to work properly:

Move_string Macro

```
move_string source,destination,num_bytes
    push    es
    mov     ax,ds
    mov     es,ax
    mov     si,offset source
    mov     di,offset destination
    mov     cx,num_bytes
    rep movs es:destination,source
    pop     es
endm
```

Convert Macro

```
convert value,base,destination
    local  table,start
    jmp    start
table db "0123456789ABCDEF"
start: mov  al,value
       xor  ah,ah
       xor  bx,bx
       div  base
       mov  bl,al
       mov  al,cs:table[bx]
       mov  destination,al
       mov  bl,ah
       mov  al,cs:table[bx]
       mov  destination[1],al
endm
```

Convert_to_binary Macro

```
convert_to_binary string,number,value
    local ten,start,calc,mult,no_mult
    jmp start
ten db 10
start: mov value,0
      xor cx,cx
      mov cl,number
      xor si,si
calc:  xor ax,ax
      mov al,string[si]
      sub al,48H
      cmp cx,2
      jl no_mult
      push cx
      dec cx
mult:  mul cs:ten
      loop mult
      pop cx
no_mult: add value,ax
      inc si
      loop calc
endm
```

Convert_date Macro

```
convert_date dir_entry
    mov dx,word ptr dir_entry[25]
    mov cl,5
    shr dl,cl
    mov dh,dir_entry[25]
    and dh,1FH
    xor cx,cx
    mov cl,dir_entry[26]
    shr cl,1
    add cx,1980
endm
```

Overview	8-4
Routines	8-5
Access	8-5
Example.....	8-5
Routine Interrupt Table	8-6
INT 5H - Print Screen Routine.....	8-7
INT 10H - Video Routines.....	8-8
Set Mode and Clear Screen	8-9
Set Cursor Type	8-9
Set Cursor Position	8-10
Read Cursor Position	8-10
Read Light Pen Position	8-10
Select Active Page Number	8-11
Scroll Active Page Up	8-11
Scroll Active Page Down	8-12
Read Attribute or Character	8-12
Write Attribute or Character.....	8-13
Write Character.....	8-13
Set Color Palette	8-14
Write Pixel	8-14
Read Pixel	8-15
Write Teletype.....	8-15

Read Current Video State	8-15
Comments	8-16
INT 11H - Equipment Check	
Routine	8-19
INT 12H - Real Mode Memory Size	
Routine	8-21
INT 13H - Disk Routines	8-22
Comments	8-23
INT 14H - Communication	
Routines	8-27
Initialize Communication Port	8-27
Send Character	8-29
Receive Character	8-29
Get Communications Port Status	8-30
INT 15H - Protected Mode Memory Size	
Routine	8-31
Comments	8-31
INT 16H - Keyboard Routines	8-32
Read Next Character	8-32
Check Keystroke	8-32
Get Current Shift Status	8-33
INT 17H - Printer Routines	8-43
Print a Character	8-43
Initialize Printer Port	8-44
Get Printer Port Status	8-44
Comments	8-44

INT 19H - Bootstrap Routine	8-45
INT 1AH - Time-Of-Day Routine	8-46
Bypassing The BIOS Routines	8-47
ROM BIOS Listing.....	8-48

Overview

This chapter describes all of the ROM BIOS service routines that are provided to perform the more low-level functions that you may need in your assembly language programs. Because these are low-level programs, they provide more direct access to the hardware than the MS-DOS routines. However, they do not provide some of the protection and conveniences that the MS-DOS routines give. Be sure to check the section on “MS-DOS Interrupts and Function Calls” to make your choice between similar MS-DOS versus BIOS routines.

Routines

BIOS routines are called with conventions that are very similar to the conventions used for calling MS-DOS routines.

ACCESS

Access to the ROM BIOS service routines is through 80286 microprocessor interrupts.

Some interrupts, like Interrupt 11H (Equipment List), perform only one routine. Others, like Interrupt 13H, have several routines that you can select.

To perform a routine, place the number of the routine in the AH register, then execute the interrupt. Use the interrupt associated with the desired routine.

EXAMPLE

To format a track on a diskette, place "05" in the AH register, then execute the interrupt associated with the disk routines (13H).

Note: You can assume that the register contents are preserved, except as noted otherwise.

You can create Macros for these routines that are similar to the macros used in Chapter 7 "System Calls" for the system calls.

Routine Interrupt Table

CODE INTERRUPT

5H PRINT SCREEN ROUTINE

10H VIDEO ROUTINES

Set mode and clear screen (0)	Read attribute or character (8)
Set cursor type (1)	Write attribute or character (9)
Set cursor position (2)	Write character (A)
Read cursor position (3)	Set color palette (B)
Read light pen position (4)	Write pixel (C)
Select active page number (5)	Read pixel (D)
Scroll active page up (6)	Write teletype (E)
Scroll active page down (7)	Read current video state (F)

11H EQUIPMENT CHECK ROUTINE

12H REAL MODE MEMORY SIZE ROUTINE

13H DISK ROUTINES

Reset diskette system (0)	Verify specified sectors (4)
Read status of diskette into AL (1)	Format a track (5)
Read sectors into memory (2)	Read DASD type (FH)
Write sectors from memory to diskette (3)	Check change line status (10H)
	Set format type (11H)

14H COMMUNICATION ROUTINES

Initialize communication port (0)
Send character (1)
Receive character (2)
Get communication port status (3)

15H PROTECTED MODE MEMORY SIZE ROUTINE

16H KEYBOARD ROUTINES

Read next character (0)
Check if keystroke available (1)
Get current shift status (2)

17H PRINTER ROUTINES

Print a character (0)
Initialize printer port (1)
Get printer port status (2)

19H BOOTSTRAP ROUTINE

1AH TIME-OF-DAY ROUTINE

Read clock (0)
Set clock (1)
Write calendar—clock (-1)
Read calendar—clock (-2)

INT 5H - Print Screen Routine

This routine prints a paper copy of the screen. This produces exactly the same result as pressing the **SHIFT** and **PrtSc** keys.

This routine works in either text or graphics modes. Unrecognizable characters are printed as blanks.

Note: No initial set-up of registers is required.

INT 10H - Video Routines

The display controller on the AT&T Personal Computer PC 6300 PLUS supports both monochrome and color displays with text or graphics. There are sixteen routines associated with the video routines interrupt.

Code	Routine
0	Set mode and clear screen
1	Set cursor type
2	Set cursor position
3	Read cursor position
4	Read light pen position
5	Select active page number
6	Scroll active page up
7	Scroll active page down
8	Read attribute or character
9	Write attribute or character
A	Write character
B	Set color palette
C	Write pixel
D	Read pixel
E	Write teletype
F	Read current video state

**SET MODE
AND CLEAR
SCREEN**

This routine establishes the mode for the display controller. Control is passed to the ROM resident Set Mode function. Set Mode initializes palette 0 as the active palette.

Input

AH = 0

Text Modes:

AL = 0 — 40X25 Monochrome mode

AL = 1 — 40X25 Color mode

AL = 2 — 80X25 Monochrome mode

AL = 3 — 80X25 Color mode

Graphics Modes:

AL = 4 — 320X200 Color mode (medium)

AL = 5 — 320X200 Monochrome mode

AL = 6 — 640X200 Monochrome mode (high)

AL = 7 — 80X25 Monochrome mode

AL = 64 — 640X400 Monochrome mode
(super high)

AL = 72 — 640X400 Monochrome mode
(tiny text)

Output

None

**SET
CURSOR
TYPE**

This routine sets the type of cursor.

Input

AH = 1

CH — Start line for cursor — Lower 5 bits

CL — End line for cursor — Lower 5 bits

Output

None

This routine sets the position of the cursor.

Input

AH = 2
BH — Page number — 0 in graphics modes
DX — Position — Row (DH), column (DL)

Output

None

READ
CURSOR
POSITION

This routine gets the position of the cursor.

Input

AH = 3
BH — Page number — 0 in graphics modes

Output

CX — Cursor start and end lines
DX — Cursor position — Row (DH), column (DL)

READ
LIGHT PEN
POSITION

This routine gets the position of the light pen.

Input

$$\text{AH} = 4$$

Output

AH = 0 — Light pen switch not triggered
 AH = 1 — Light pen value obtained

BX = 0 through 639 — Pixel column (high resolution)
 BX = 0 through 319 — Pixel column (medium resolution)

CH = 0 through 199 — Raster line
 DX — Light pen position — Row (DH), column (DL)

**SELECT
ACTIVE
PAGE
NUMBER**

This routine selects a new active page.

Input

AH = 5

AL = 0 through 7 — Page number
(modes 0 & 1)

AL = 0 through 3 — Page number
(modes 2 & 3)

Output

None

**SCROLL
ACTIVE
PAGE UP**

This routine selects the next page up.

Input

AH = 6

AL — Number of lines blanked at bottom
(0 means blank entire window)

BH — Attribute — Used on blank line

CX — Upper left corner — Row (CH),
column (CL)

DX — Lower right corner — Row (DH),
column (DL)

Output

None

**SCROLL
ACTIVE
PAGE DOWN**

This routine selects the next page down.

Input

AH = 7

AL — Number of lines blanked at top
(0 means blank entire window)

BH — Attribute — Used on blank line

CX — Upper left corner — Row (CH),
column (CL)

DX — Lower right corner — Row (DH),
column (DL)

Output

None

**READ
ATTRIBUTE
OR
CHARACTER**

This routine gets the value of the character at
the current cursor position.

Input

AH = 8

BH — Current display page

Note: Written at current cursor position.

Output

AH — Attribute of character

AL — Character

**WRITE
ATTRIBUTE
OR
CHARACTER**

This routine sets the value of the character at the current cursor position.

Input

AH = 9

AL — Character

BH — Current display page

BL — Attribute of character (text modes)

BL — Color of character (graphics modes)

CX — Number of characters

Note: Written at current cursor position.

Output

None

**WRITE
CHARACTER**

This routine displays the character value, in the AL register, at the current cursor position.

Input

AH = 0AH

AL — Character

BH — Current display page

CX — Number of characters

Note: Written at current cursor position.

Output

None

**SET COLOR
PALETTE**

This routine sets the color values in one of the palettes. Also, can be used to reset palettes to default values.

Input

AH = 0BH

BH = 0 or 1 — Palette

0 = green/red/yellow

1 = cyan/magenta/white

BL — Color — See table

0 = black

8 = gray

1 = blue

9 = light blue

2 = green

A = light green

3 = cyan

B = light cyan

4 = red

C = light red

5 = magenta

D = light magenta

6 = brown

E = yellow

7 = white

F = high intensity white

Output

None

**WRITE
PIXEL**

This routine displays a pixel to a specified color and position.

Input

AH = 0CH

AL — Color

CX — Column number

DX — Row number

Note: If bit 7 of AL is set (=1), the color value is exclusively OR'd with the current value of the pixel.

Output

None

**READ
PIXEL**

This routine gets the value of a specified pixel position.

Input

AH = 0DH
CX — Column number
DX — Row number

Output

AL — Pixel value

WRITE TELETYPE**Input**

AH = 0EH
AL — Character
BH — Display page
BL — Foreground color (Graphics modes)

Note: Screen width is set by previous mode.

Output

None

**READ
CURRENT
VIDEO
STATE**

This routine gets the current video state.

Input

AH = FH
AL — Current mode
AH — Number of columns
BH — Current active display page

Output

AH = 1 through 132 — Number of columns
AL — Current mode
BH — Current display page

COMMENTS

Monochrome Text Mode

The monochrome text modes are:

- mode 0 — 40×25 characters
- mode 2 — 80×25 characters.

Your program dictates what is displayed on the screen by what is stored in these locations. Display storage in memory starts at the top left screen position and continues on a line-by-line basis to the bottom right position.

For each screen position, there are two bytes saved in memory. The first byte is the character code to be displayed. The second byte is the attribute code that specifies how the character is to be displayed. This attribute byte specifies brightness, underlining, and blinking.

Color Text Modes

The color text modes are:

- mode 1 — 40×25 color mode
- mode 3 — 80×25 color.

Memory usage for the color text modes is similar to the method used for monochrome text. Two bytes of memory are used for each character position: the first is the character code and the second is the attribute byte. The attribute byte specifies blinking, brightness, and color.

The display memory maps to the character position exactly as it does in monochrome text mode. The big difference in color is that eight pages of memory are used to build up to eight separate screens (pages). Only one page is

active at any time, but you can switch the active page number and thereby display screens very rapidly.

The display pages are numbered 0 - 7 for 40×25 mode and 0 - 3 for 80×25 mode. For 40-column mode, new pages occur at 2K byte intervals; for 80-column, at 4K byte intervals. A total of 32K bytes of memory is used.

Color Graphics Mode

The color graphics mode is mode 4 — medium resolution (320×210) color graphics. For any color display, you can use up to four colors. You select from one of two “palettes,” each of which provides three colors. You select a “background” color to be used as the fourth color.

Palette 0 contains green, yellow, and red. Palette 1 contains cyan (light blue), magenta (deep purplish red), and white.

320 pixels can be displayed on each of 200 lines. Each line takes 80 bytes or 640 bits of display memory. Each color pixel uses two bits of memory. Since two bits give you four possible combinations, you specify either the background color or one of the three colors in the current palette.

High Resolution Monochrome Graphics

The memory for high-resolution monochrome graphics (640×200 pixels), mode 6, is handled similarly to color/graphics.

The display memory for high-resolution monochrome graphics starts at B8000H. It is divided into two 4K blocks. Each line has 640 pixels. Each bit of display memory turns one pixel either on or off. The even lines of pixels start at location B8000H; the odd lines of pixels start at C8000H.

Monochrome Graphics

Super-high resolution monochrome graphics (640×400), mode 64, uses 32K bytes of memory. One bit represents one pixel. Memory is divided into four 8K bytes sections as follows:

Memory Location	Number
B8000	0, 4, 8, ...
BA000	2, 6, 10, ...
BC000	1, 5, 9, ...
BE000	3, 7, 11, ...

Character Handling

If your program displays characters to the screen while in graphics modes, the characters are formed from a character generator image that is maintained in the ROM. However, only the first 128 characters are encoded there.

To create your own characters or symbols, either for the purposes of doing character graphics or implementing a foreign alphabet, you must set up a table of code points for up to 128 new characters or symbols. Also, initialize the pointer at interrupt 1F (address 0007CH) to point the table. These characters can then be accessed by referring to codes 128 through 255. Codes 0 through 127 are the ASCII set maintained in ROM.

In text mode, if you send more characters to be written than will fit on one line, characters automatically wrap around to the beginning to the next line.

In graphics mode, the character-handling routines only produce correct results for characters contained on the same row (continuation to succeeding lines does not work.)

INT 11H - Equipment Check Routine

You can use this routine to obtain a list of the optional equipment attached to your system.

INPUT None

OUTPUT AX = Equipment list — In this format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	# P			G		# C			# D	M			M	2		D
	R			A		O			I	O			E	8		I
	N		-	M		M		-	S	D			M	7		S
Cnts	T			E		M			K	E						K

#PRNT - # of Printer Adapters

These two bits indicate the number of printer adapters on this system. Contents can be 0 through 3.

GAME - Game Adapter

This bit indicates whether or not the game adapter is attached. The bit set (=1) indicates attached.

#COMM - # of Communication Adapters

These three bits indicate the number of communication adapters on this system. Contents can be 0 through 7.

#DISK - # of Disk Drives

These two bits indicate the number of disk drives on the system. The actual number is found by adding 1 to the contents. That is, if the #DISK field is 3 this indicates four drives.

MODE - Starting Video Mode

These two bits indicate the starting video mode. The various modes and their codes are as follows:

- 1 — Monochrome graphics with 40-columns
- 2 — Monochrome graphics with 80-columns
- 3 — Monochrome text

MEM - Memory on system

These two bits indicate the amount of memory on the system board. "11" (3H) indicates 64K bytes, which is always set.

287 - 80287 Numeric Processor

Extension

This bit indicates whether or not the 80287 numeric processor extension is attached. The bit set (=1) indicates the presence of the processor.

DISK - Disk Drives

This bit indicates whether or not disk drives are attached. The bit set (=1) indicates attached.

INT 12H - Real Mode Memory Size Routine

This routine indicates the total amount of memory in the address space, up to one megabyte.

The BIOS reads the contents of the "memory-size" location in RAM (address 413H) previously set by the power-up diagnostics. The value is returned in the AX register.

Note: The value returned is "real mode" memory only. That is, the contiguous memory located below 1 megabyte.

INPUT None

OUTPUT AX = 0 through 0280H (640 decimal) — Number of 1K memory blocks

Note: No initial set-up of registers is required.

INT 13H - Disk Routines

There are six routines associated with the disk routines interrupt.

Code	Routine
0	Reset diskette system
1	Read status of diskette into AL
2	Read sectors into memory
3	Write sectors from memory to diskette
4	Verify specified sectors
5	Format a track
F	Read DASD type
10H	Check change line status
11H	Set format type

INPUT AH = 0 through 11H — Routine code

Provide the following in addition to AH:

For AH = 0 through 5:

AL = 1 through 15 — Number of sectors

For AH = 17:

AL = 1 through 8 — Media type as follows:

1 = 250K bits/second; 360K byte low
density drive

2 = 300K bits/second; 360K byte high
density drive

3 = 500K bits/second; 1.2M byte high
density drive

CH = 0 through 39 — Track number
CL = 1 through 9 — Sector number
DH = 0 or 1 — Head number
DL = 0 through 3 — Drive number
BX — Starting address of buffer
ES — Segment of buffer

OUTPUT

Status Code:

AH = 01 — Bad command
AH = 02 — Address mark not found
AH = 03 — Write requested on protected disk
AH = 04 — Requested sector not found
AH = 06 — Media change
AH = 08 — DMA overrun
AH = 09 — DMA transfer crossed a 64K bytes boundary
AH = 10 — Read data error detected by CRC
AH = 20 — Diskette controller chip failed
AH = 40 — Seek to desired track failed
AH = 80 — Device timeout

CY = 0 — Successful operation
CY = 1 — Unsuccessful operation
(AH has details)

Fixed Disk

The fixed disk, if present, is also accessed by using interrupt 13H. A front-end -s added which checks the drive number in the DL register. The DL equals 80H command is intended for the fixed disk.

80H = 0 — Fixed disk (first)

81H = 1 — Fixed disk (second)

80 selects the first fixed disk (usually Drive C).

81 selects the second fixed disk (usually Drive D).

The commands are as follows:

AH = 0 — Reset Controller

AH = 1 — Read status of last operation

AH = 2 — Read sector(s)

AH = 3 — Write sector(s)

AH = 4 — Verify sector(s)

AH = 5 — Format track

AH = 6 — Format bad track

AH = 7 — Format drive

AH = 8 — Read drive parameters

AH = 9 — Initialize drive parameters

AH = AH — Read long

AH = BH — Write long

Note: A “Read long” or “Write long” transfers 512 bytes of data plus 4 bytes of ECC.

AH = CH — Seek
AH = DH — Reset
AH = EH — Read sector buffer
AH = FH — Write sector buffer
AH = 10H — Test drive ready
AH = 11H — Recalibrate
AH = 12H — Controller RAM diagnostics
AH = 13H — Drive diagnostic
AH = 14H — Controller diagnostic

After the command the AH register contains the completion code as follows:

AH = 01H — Bad command
AH = 02H — Adrs mark not found
AH = 05H — Reset failed
AH = 07H — Initialize drive failed
AH = 09H — Attempt to DMA access
 64K byte boundary
AH = 0BH — Bad track
AH = 10H — Uncorrectable data error
AH = 11H — ECC correctable data error
AH = 20H — Controller failed self test
AH = 40H — Seek failed
AH = 64H — Sector not found
AH = 80H — No response from device
AH = BBH — Undefined error
AH = FFH — Read status failed

Other registers contain as follows:

AL — Number of sectors

Note: When a format command is issued, the AL register indicates the interleave value.

CH = 0 through 611 — Cylinder number
CL = 1 through 17 — Sector number
DH = 0 or 1 — Head number
DL = 0 through 3 — Drive number
BX — Starting address of buffer
ES — Segment of buffer

When the cylinder number is greater than 255, the upper two bits of the cylinder number are placed in the upper two bits of the sector register (CL).

INT 14H - Communication Routines

This set of routines performs serial RS232C communications through the communications port. These routines are appropriate when a polling technique is used; this is not interrupt-driven I/O.

Code Routine

0	Initialize communication port
1	Send character
2	Receive character
3	Get communication port status

INITIALIZE COMMUNICATION PORT This routine initializes the communication port.

Input AH = 0
 AL — Parameter
 DX = 0 or 1 — Communications adapter board

Output Registers are set the same as for GET COMMUNICATIONS PORT Routine (3).

Bit	7	6	5	4	3	2	1	0
Cnts	BAUD			PARITY		ST	WORD	

BAUD

These three bits indicate the baud rate:

Code	Baud Rate
000	110
001	150
010	300
011	600
100	1200
101	2400
110	4800
111	9600

PARITY

These two bits indicate the parity:

Code	Parity
00	None
01	Odd
02	Even

ST

This bit indicates the number of start/stop bits:

Code	Number
0	1
1	2

WORD

These two bits indicate the size of the data word:

Code	Size
10	7 bits
11	8 bits

**SEND
CHARACTER**

This routine sends a character to the communication port.

Input

AH = 1

AL = Character to be sent

DX = 0 or 1 — Communications adapter board

Output

AH(bit 7) = 0 — Operation was unsuccessful

**RECEIVE
CHARACTER**

This routine gets a character from the communication port.

Input

AH = 2

Output

AH = 0 — Operation was successful

AH — Current line status — Registers are set
the same as for Code 3.

AL — Received character

GET COMMUNICATIONS PORT STATUS This routine gets the communications port status.

Input AH = 3

Output AH — Line control status — In this format:

Bit	Contents
7	Timeout
6	Transmission shift register empty
5	Transmission holding register empty
4	Break detect
3	Framing error
2	Parity error
1	Overrun error
0	Data ready

AL — Modem status — In this format:

Bit	Contents
7	Received line signal detect
6	Ring detect
5	Data set ready
4	Clear to send
3	Delta receive line signal detect
2	Trailing edge ring detected
1	Delta data set ready
0	Delta clear to send

INT 15H - Protected Mode Memory Size Routine

INPUT	AH = 88H
OUTPUT	AX = 0 through 15360 — Number of memory blocks above 1 Megabyte (Protected Mode RAM).
COMMENTS	The BIOS reads the contents of the p_memsize location in RAM (address 4CAH) and places the value in the AX register.

INT 16H - Keyboard Routines

There are three routines associated with the keyboard routines interrupt.

Code	Routine
0	Read next character
1	Check if keystroke available
2	Get current shift status

READ NEXT CHARACTER

This routine will not return execution to the calling program until it has a keystroke to report.

Input

AH = 0

Output

AH — Raw scan code

AL — ASCII translated code or AH — Scan code

AL = 00H

If the key does not have an ASCII code (i.e. **F1** key), then:

AL = 08H

AH — Scan code

CHECK KEYSTROKE

This routine is used to check to see if a keystroke has been entered. Use this routine to continue processing whether or not a key has been pressed.

Input AH = 1

Output AX — Character code
 Z flag — Status ("0" if available)

Note: The character code remains in the keyboard buffer. Therefore, execution of the **Read next character** routine will return the same value.

**GET
CURRENT
SHIFT
STATUS** This routine gets the current shift status.

Input AH = 2

Output AL — Current shift status — In this format:

Byte	Bit	Subject matter	Meaning, when bit is 1
1	1	Insert	state active
1	2	Caps-Lock	state active
1	3	Num-Lock	state active
1	4	Scroll-Lock	state active
1	5	Alt shift	key depressed
1	6	Ctrl shift	key depressed
1	7	left-hand shift	key depressed
1	8	right-hand shift	key depressed
2	1	Insert	key depressed
2	2	Caps-Lock	key depressed
2	3	Num-Lock	key depressed
2	4	Scroll-Lock	key depressed
2	5	hold state	state active (from Ctrl-Num-Lock)
2	6	(not used)	
2	7	(not used)	
2	8	(not used)	

CHARACTER CODES

KEY #	LOWER CASE	UPPER CASE	CTRL	ALT
1	ESC	ESC	ESC	-1
2	1	1/2	-1	*
3	2	@	(000)*	*
4	3	#	-1	*
5	4	\$	-1	*
6	5	%	-1	*
7	6	-	RS (030)	*
8	7	&	-1	*
9	8	*	-1	*
10	9	(-1	*
11	0)	-1	*
12	-	—	US (031)	*
13	=	+	-1	*
14	BACKSPACE (008)	BACKSPACE (008)	DEL (127)	-1
15	—>!(009)	!<—*	-1	-1
16	q	Q	DC1 (017)	*
17	w	W	ETB (023)	*
18	e	E	ENQ (005)	*
19	r	R	DC2 (018)	*
20	t	T	DC4 (020)	*
21	y	Y	EM (025)	*
22	u	U	NAK (021)	I
23	i	I	HT (009)	*
24	o	O	SI (015)	*
25	p	P	DLE (016)	*
26	[{	ESC (027)	-1
27]	}	GS (029)	-1
28	CR	CR	LF (010)	-1
29 CTRL	-1	-1	-1	-1
30	a	A	SOH (001)	*
31	s	S	DC3 (019)	*
32	d	D	EOT (004)	*
33	f	F	ACK (006)	*
34	g	G	BEL (007)	*
35	h	H	BS (008)	*
36	j	J	LF (010)	*
37	k	K	VT (011)	*

CHARACTER CODES (Cont'd)

KEY #	LOWER CASE	UPPER CASE	CTRL	ALT
38	l	L	FF (012)	*
39	;	:	-1	-1
40	"	"	-1	-1
41	'	'	-1	-1
42 SHIFT	-1	-1	-1	-1
43		!	FS (028)	-1
44	z	Z	SUB (026)	*
45	x	X	CAN (024)	*
46	c	C	ETX (003)	*
47	v	V	SYN (022)	*
48	b	B	STX (002)	*
49	n	N	SO (014)	*
50	m	M	CR (013)	*
51	'	<	-1	-1
52	.	>	-1	-1
53	/	?	-1	-1
54SHIFT	-1	-1	-1	-1
55	*	**	*	-1
56ATL	-1	-1	-1	-1
57	SP	SP	SP	SP
58CAPS LOCK	-1	-1	-1	-1
59	NUL*	NUL*	NUL*	NUL*
60	NUL*	NUL*	NUL*	NUL*
61	NUL*	NUL*	NUL*	NUL*
62	NUL*	NUL*	NUL*	NUL*
63	NUL*	NUL*	NUL*	NUL*
64	NUL*	NUL*	NUL*	NUL*
65	NUL*	NUL*	NUL*	NUL*
66	NUL*	NUL*	NUL*	NUL*
67	NUL*	NUL*	NUL*	NUL*
68	NUL*	NUL*	NUL*	NUL*
69NUM LOCK	-1	-1	Pause	-1
70SCROLL LOCK	-1	-1	Break	-1

* Note 1: Refer to Extended Codes.

CHARACTER CODES (Cont'd)

Keys 71-83 have meaning only in base case, in NUM-LOCK (or shifted) states, or in CTRL state. It should be noted that the shift key temporarily reverses the current NUM-LOCK state.

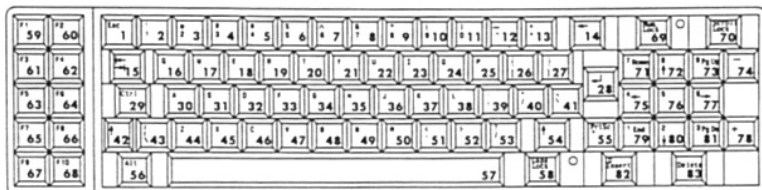
KEY #	NUM LOCK	LOWER CASE	ALT	CTRL
71	7	Home*	*	Clear Screen
72	8	~*	*	-1
73	9	Page Up*	*	Top of Text & Home
74	-	-	-1	-1
75	4	<—*	*	Reverse Word*
76	5	-1	*	-1
77	6	m>*	*	Adv word*
78	+	+	-1	-1
79	1	End*	*	Erase to EOL *
80	2	*	*	-1
81	3	Page Down*	*	Erase to EOS *
82	0	INS	*	-1
83		DEL *,**	**	**

* Note 1: Refer to Extended Codes.

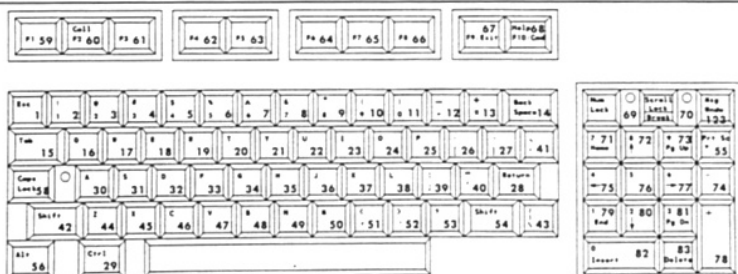
CHARACTER CODES (Cont'd)

Extended Codes

SECOND CODE	FUNCTION
3	NUL Character
15	<—
16-25	ALT Q,W,E,R,T,Y,U,I,O,P
30-38	ALT A,S,D,F,G,H,J,K,L
44-50	ALT Z,X,C,V,B,N,M
59-68	F1-F10 Function Keys Base Case
71	Home
72	
73	Page Up & Home Cursor
75	<—
77	—>
79	End
80	
81	Page Down & Home Cursor
82	INS
83	DEL
84-93	F11-F20 (Upper Case F1-F10)
94-103	F21-F30 (CTRL F1-F10)
104-113	F31-F40 (ALT F1-F10)
114	CTRL PRTSC (Start/Stop Echo to Printer) Key 55
115	CTRL <— Reverse Word
116	CTRL —> Advance Word
117	CTRL END Erase EOL
118	CTRL PG DN Erase EOS
119	CTRL HOME Clear Screen and home
120-131	ATL 1,2,3,4,5,6,7,8,9,0,-,(Keys 2-13)
132	CTRL PG UP TOP 25 Lines of Text & Home Cursor



Model 301 Keyboard



Model 302 Keyboard

CHARACTER CODES

ASCII Decimal	Value Hex	Character	Control Character	ASCII Decimal	Value Hex	Character
000	00	(null)	NUL	032	20	(space)
001	01	☺	SOH	033	21	!
002	02	☹	STX	034	22	"
003	03	♥	ETX	035	23	#
004	04	♦	EOT	036	24	\$
005	05	♣	ENQ	037	25	%
006	06	♠	ACK	038	26	&
007	07	(beep)	BEL	039	27	'
008	08	▣	BS	040	28	(
009	09	(tab)	HT	041	29)
010	0A	(line feed)	LF	042	2A	*
011	0B	(home)	VT	043	2B	+
012	0C	(form feed)	FF	044	2C	,
013	0D	(carriage return)	CR	045	2D	.
014	0E	♪	SO	046	2E	.
015	0F	☼	SI	047	2F	/
016	10	▶	DLE	048	30	0
017	11	◀	DC1	049	31	1
018	12	↕	DC2	050	32	2
019	13	!!	DC3	051	33	3
020	14	π	DC4	052	34	4
021	15	§	NAK	053	35	5
022	16	▬	SYN	054	36	6
023	17	↕	ETB	055	37	7
024	18	↑	CAN	056	38	8
025	19	↓	EM	057	39	9
026	1A	→	SUB	058	3A	:
027	1B	←	ESC	059	3B	;
028	1C	(cursor right)	FS	060	3C	<
029	1D	(cursor left)	GS	061	3D	=
030	1E	(cursor up)	RS	062	3E	>
031	1F	(cursor down)	US	063	3F	?

CHARACTER CODES (Cont'd)

ASCII Decimal	Value Hex	Character	Control Character	ASCII Decimal	Value Hex	Character
064	40	@		096	60	`
065	41	A		097	61	a
066	42	B		098	62	b
067	43	C		099	63	c
068	44	D		100	64	d
069	45	E		101	65	e
070	46	F		102	66	f
071	47	G		103	67	g
072	48	H		104	68	h
073	49	I		105	69	i
074	4A	J		106	6A	j
075	4B	K		107	6B	k
076	4C	L		108	6C	l
077	4D	M		109	6D	m
078	4E	N		110	6E	n
079	4F	O		111	6F	o
080	50	P		112	70	p
081	51	Q		113	71	q
082	52	R		114	72	r
083	53	S		115	73	s
084	54	T		116	74	t
085	55	U		117	75	u
086	56	V		118	76	v
087	57	W		119	77	w
088	58	X		120	78	x
089	59	Y		121	79	y
090	5A	Z		122	7A	z
091	5B	[123	7B	{
092	5C	\		124	7C	
093	5D]		125	7D	}
094	5E	^		126	7E	~
095	5F	_		127	7F	␣

CHARACTER CODES (Cont'd)

ASCII Decimal	Value Hex	Character	ASCII Decimal	Value Hex	Character
128	80	Ç	160	A0	á
129	81	ü	161	A1	í
130	82	é	162	A2	ó
131	83	â	163	A3	ú
132	84	ä	164	A4	ñ
133	85	à	165	A5	N
134	86	â	166	A6	a
135	87	ç	167	A7	o
136	88	ê	168	A8	z
137	89	ë	169	A9]
138	8A	è	170	AA	[
139	8B	ÿ	171	AB	½
140	8C	î	172	AC	¼
141	8D	ì	173	AD	i
142	8E	À	174	AE	«
143	8F	Å	175	AF	»
144	90	É	176	B0	•••
145	91	æ	177	B1	•••
146	92	Æ	178	B2	•••
147	93	ô	179	B3	
148	94	ö	180	B4	┌
149	95	ò	181	B5	┐
150	96	û	182	B6	└
151	97	ù	183	B7	┘
152	98	ÿ	184	B8	┘
153	99	Ö	185	B9	┘
154	9A	Ü	186	BA	=
155	9B	ç	187	BB	┘
156	9C	£	188	BC	┘
157	9D	¥	189	BD	┘
158	9E	Pt	190	BE	┘
159	9F	f	191	BF	┘

CHARACTER CODES (Cont'd)

ASCII Decimal	Value Hex	Character	ASCII Decimal	Value Hex	Character
192	C0		224	E0	
193	C1		225	E1	
194	C2		226	E2	
195	C3		227	E3	
196	C4		228	E4	
197	C5		229	E5	
198	C6		230	E6	
199	C7		231	E7	
200	C8		232	E8	
201	C9		233	E9	
202	CA		234	EA	
203	CB		235	EB	
204	CC		236	EC	
205	CD		237	ED	
206	CE		238	EE	
207	CF		239	EF	
208	D0		240	F0	
209	D1	¡	241	F1	¡
210	D2	¢	242	F2	¢
211	D3	£	243	F3	£
212	D4	¤	244	F4	¤
213	D5	¥	245	F5	¥
214	D6	¦	246	F6	¦
215	D7	§	247	F7	§
216	D8	¨	248	F8	¨
217	D9	©	249	F9	©
218	DA		250	FA	
219	DB		251	FB	
220	DC		252	FC	
221	DD		253	FD	
222	DE		254	FE	
223	DF		255	FF	(blank)

INT 17H - Printer Routines

This set of BIOS routines communicates with the printer. There are three routines associated with the keyboard routines interrupt. For all of these printer services, set (DX) equal to the printer to be used (0, 1, or 2).

Code	Routine
0	Print a character
1	Initialize printer port
2	Get printer port status

INPUT

AH = 0 through 2 — Routine code
DX = 0 through 1 — Printer
AL — Character to be printed (Code 0 only)

PRINT A CHARACTER

This routine sends a character to the printer port.

Input

AH = 0
DX = 0 through 1 — Printer
AL — Character to be printed (Code 0 only)

Output

AH = 1 — Timeout (not printed)

**INITIALIZE
PRINTER
PORT** This routine initializes the printer port.

Input AH = 1
 DX = 0 through 1 — Printer

Output AH — Printer status — In this format:

Bit	Contents
7	Busy
6	Acknowledge
5	Out of paper
4	Selected
3	I/O error
2	Not used
1	Not used
0	Timeout

Note: Status values 90H and 10H indicate the printer is connected. Status value 38H indicates printer is disconnected.

**GET
PRINTER
PORT
STATUS** This routine gets the printer port status.

Input AH = 2

Output AH — Printer status — Same format as Code 1

COMMENTS None

INT 19H - Bootstrap Routine

This routine resets (bootstraps) the system to the initial power-up conditions (after the ROM power-up diagnostics).

The system bootstraps itself in much the same way as it does with the **CTRL** **ALT** **DELETE** which causes diagnostics to be run, whereas Interrupt 19H causes an immediate system load.

Note: No initial set-up of registers is required.

INT 1AH - Time-Of-Day Routine

This set of BIOS routines communicates with the real time clock and calendar clock hardware.

Note: No initial set-up of registers is required.

Code	Routine
0	Read the real time clock
1	Set the real time clock
-1	Set the calendar clock
-2	Read the calendar clock

AH = 0, 1, -1, or -2

BX = 0 through 0B69H — Number of days
since 1/1/84

CH = 0 through 23 — Hour

CL = 0 through 59 — Minute

DH = 0 through 59 — Seconds

DL = 0 through 99 — Tenths of a second

Output

AH = -1 — Date/Time error

Bypassing The BIOS Routines

This section explains how you can either replace one of the BIOS service routines with a program of your own, or add a "front-end" so that you perform some preprocessing immediately prior to using a particular BIOS routine.

When the system is powered on, low memory is initialized with the addresses of all of the BIOS interrupt routines. To replace a BIOS routine, change the address in the interrupt table to the address of the code you want to execute in place of the BIOS code.

To perform preprocessing before handing execution on to the BIOS code:

- 1 Replace the address of the BIOS routine with the address of your program
- 2 Transfer execution to the BIOS routine at the end of your program.

Also, there are system calls to aid with redirecting the interrupt vectors.

ROM BIOS Listing

The ROM BIOS listing and index are included in a separate manual (*ROM BIOS Listings, AT&T Personal Computer 6300 PLUS*). It is not included with the *System Programmer's Guide* but can be purchased separately to supplement your PC 6300 PLUS library.

Overview	9-5
Interested Audience	9-5
What is an MS-DOS Device Driver?	9-5
How to Create a Device Driver	9-7
How to Install a Device Driver	9-7
Background	9-8
Programmable Devices	9-8
I/O Ports	9-9
I/O Instructions	9-9
Interrupts	9-10
Interrupt Devices	9-12
Block Diagrams	9-12
Device Headers	9-15
Format	9-15
Request Header	9-20
Format	9-20
Record Length	9-20
Unit Code	9-20
Command Code	9-21
Status	9-33
Reserved	9-34

Hardware Controller Documentation	9-35
Asynchronous Communications Element (ACE)	9-36
Functions	9-37
Registers	9-37
Sequencing and Timing	9-42
Setting the Baud Rate	9-42
Break Control Feature	9-43
Interrupt Priority	9-44
Communications Manager Interface	9-45
Functions	9-46
Comments	9-50
Command Protocol	9-55
Display Controller	9-66
Functions	9-68
Registers	9-68
Modes	9-74
Sequencing and Timing	9-80
DMA Controller	9-82
Functions	9-84
Registers	9-86
Sequencing and Timing	9-91
Floppy Diskette Controller (FDC)	9-92
Registers	9-93
Sequencing and Timing	9-114
Comments	9-114
Interrupts	9-117

Hard Disk Unit Controller	9-118
Registers	9-118
Functions	9-121
Sequencing and Timing	9-134
Keyboard Interface	9-139
Functions	9-140
Registers	9-140
Sequencing and Timing	9-142
Keyboard Commands/Responses	9-142
Mouse (Optional)	9-144
Mouse Operation	9-144
Parallel Printer Interface	9-147
Functions	9-148
Registers	9-148
Sequencing and Timing	9-149
Programmable Interrupt Controller (PIC)	9-151
Registers	9-152
Sequencing and Timing	9-159
Programmable Interval Timer	9-161
Functions	9-163
Registers	9-163
Sequencing and Timing	9-164

Real-Time Clock And Calendar	9-166
Functions	9-167
Registers	9-167
Sequencing and Timing	9-171
Speaker	9-172
Registers	9-172
Sequencing and Timing	9-174
Switching Between Real and Protected Modes	9-175

Overview

INTERESTED AUDIENCE

This Chapter contains information on how to write device drivers. You may not use it often since many input and output capabilities are implemented by the BIOS routines discussed in Chapter 8. BIOS routines allow you to do general input and output without a detailed understanding of the hardware and shield your program from hardware changes.

However, there are times when BIOS routines do not perform the necessary function or do so in an inefficient fashion. Then your own driver is necessary. Implementation of operating systems, of high-speed graphics packages, and of unusual keyboard mapping are examples of software which require specialized drivers.

WHAT IS AN MS-DOS DEVICE DRIVER?

A device driver is binary code which manipulates hardware in the MS-DOS environment. A special header at the beginning identifies it as a driver, defines the strategy and interrupt entry points, and describes various attributes of the device. The file must have an origin of zero.

There are two kinds of devices:

- Character devices
- Block devices.

Character Devices

Character devices perform serial character I/O like CONSOLE, AUXILIARY, and PRINTER. These devices are named, and users open channels to do I/O to them.

Block Devices

Block devices are the disk drivers on the system. They perform random I/O in pieces called blocks. This is usually the physical sector size.

Drive letters are assigned to device drivers based on their ordering in the CONFIG.SYS file. Starting with the letter 'A', each device driver is assigned as many consecutive alphabetic characters as the driver has units. The theoretical limit is 63, but after 26 the drive letters are non-alphabetic (such as "[") and "^").

Character devices cannot define multiple units because they have only one name.

HOW TO CREATE A DEVICE DRIVER

To create a device driver, write a binary file with a Device Header at the beginning of the file. The code originates at 0.

MS-DOS always processes installable device drivers before handling the default devices. To install a new CON device, simply name the device CON. Remember to set the standard input device and standard output device bits in the attribute word on the new CON device. The scan of the device list stops on the first match, so the installable device driver takes precedence.

MS-DOS installs the driver anywhere in memory; therefore, be careful with memory references. Do not expect the driver to be loaded in the same place.

HOW TO INSTALL A DEVICE DRIVER

MS-DOS allows new device drivers to be installed dynamically at boot time. This is accomplished by INIT code in the BIOS which processes the CONFIG.SYS file.

Background

The following is provided for background. You should become familiar with these areas before using device drivers. Also, you may want to read and refer to Chapter 5 "Memory and Disk Allocation" when working with device drivers.

PROGRAM- MABLE DEVICES

Programmable devices are character and block devices which are available for use with device drivers. This is a list of programmable devices.

Type	Device
INS 8250A	Asynchronous Communications Element
INTEL 8237A	DMA Controller
NEC uPD765	Floppy Diskette Controller
WD WD1002-WX2	Hard Disk Unit Controller
INTEL 8041	Keyboard Interface Parallel Printer Interface
INTEL 8259A	Programmable Interrupt Controller
INTEL 8254	Programmable Interval Timer
National Semiconductor MM58274A	Real-Time Clock and Calendar
	Speaker Interface
MOTO 6845	Display Controller

I/O PORTS

I/O ports are addressable locations (like memory) which allow you to read or write information to or from an I/O device. Each I/O device has an associated address location for input and/or output purposes. The term I/O device refers to hardware under the control of the CPU. Each device requires a different device header, but they all need some combination of control information, status information, and data.

I/O ports are 16-bit-wide addresses as opposed to the 20-bit-wide memory addresses.

**I/O
INSTRUCTIONS**

I/O port addressing is distinguished from memory addressing via I/O instructions. I/O instructions are translated into control signals which define the direction and path of the data.

Port addressing can be specified in one of two ways:

- Fixed Addressing
- Variable Addressing.

**Fixed
Addressing**

In the fixed method, the I/O port address is specified in the instruction. For example, to send the data in the AL register to the device associated with address 20H, type:

```
OUT 20H,AL
```

Fixed addressing can only be used with 8-bit-wide I/O ports.

**Variable
Addressing**

The variable method allows 16-bit port addresses to be specified. For example, to send the data in the AL register to the device associated with the address in the DX register; load the I/O port address into the DX register and type:

```
MOV DX,03F2H
OUT DX,AL
```

Since the DX register is used, a 16-bit-wide I/O port can be addressed.

INTERRUPTS

External devices do not need the CPU's attention all the time. When they need servicing, they ask for it either by interrupting or by setting a polled flag. External interrupts are ignored when the CLI instruction has cleared the interrupt enable flag and recognized when the STI instruction sets the flag.

The first 1024 bytes of memory contain an interrupt table. This table has 255 interrupt pointers defining the start address of interrupt service routines.

This is the pointer format.

INTERRUPT POINTER

IP
CS

When the CPU recognizes the interrupt, an 8-bit interrupt type identifies the device. The interrupt type is an index into the table of pointers. To obtain the interrupt pointer address, the type is multiplied by four. The CPU saves the flag register on the stack, disables interrupts and single step mode, and saves the CS and IP registers on the stack. Then the interrupt pointer is loaded into IP and CS, and control is transferred to the interrupt service routine. The stack looks like this when the service routine gets control.

SYSTEM STACK

IP
CS
FLAGS

When the service routine begins, interrupts are disabled. Depending on the nature of the application, interrupts can be enabled immediately or just prior to releasing control.

The service routine must preserve the value of all internal registers. Therefore, it saves the registers it uses on the stack. Before running, it restores these registers from the stack.

To return control to the interrupt program, the service routine executes a IRET. All the information necessary to do this was carefully placed on the stack.

INTERRUPT DEVICES

Type	Device
INS 8250A	Asynchronous Communications Element
NEC uPD765	Floppy Diskette Controller
WD WD1002-WX2	Hard Disk Unit Controller
INTEL 8041	Keyboard Interface Parallel Printer Interface
INTEL 8254	Programmable Interval Timer

Clock Device

One of the special enhancements for the PC 6300 PLUS is the battery-backed MM58274A clock-calendar chip and related driver. This chip is integrated into the system as the CLOCK device and is accessed with the TIME and DATE command.

This CLOCK device defines and performs functions like any other character device. When a read or write to this device occurs, exactly 6 bytes are transferred. The first two bytes are the count of days since 1-1-84. The third byte is minutes, the fourth, hours, the fifth, hundredths of seconds, and the sixth, seconds.

Reading the CLOCK device sets the date and time; writing to it sets the date and time.

BLOCK DIAGRAMS

Block diagrams are pictorial descriptions of the electrical connection between the CPU, the interface, and the external device. In general, the CPU's bus signals appear on the left. The middle of the diagram describes the internals of the interface. The right side of the diagram describes the electrical interface. This physically connects the interface to the external device. The connection is often through a dual inline package (DIP) of pins. Each pin carries one signal.

Individual signals are represented by lines. Busses are shown on double lines. Each of these have arrows which indicate the direction of the data. Often they are bidirectional.

There are several common CPU control bus signals. Their mnemonics and definitions follow:

- **A0-A19**
These are the lines used to transmit the address of memory or the address of an I/O port. Only A0-A15 are used for I/O addressing.
- **CS**
This signal selects the chip. No reading or writing will occur unless the device is selected.
- **D0-D7**
These are the bidirectional data lines used to exchange information with a memory location or an I/O port. D7 is the most significant bit.
- **INT0-INT7**
These are the priority interrupt request lines. See the description of the Interrupt Controller.
- **IORD**
This signal indicates that an input port address has been placed on the address bus. The data at the specified port is to be placed on the data bus.

- **IOWR**

This signal indicates that an output port address has been placed on the address bus and the data has been placed on the data bus to be output to the specified port.

- **RESET**

This signal resets the system to a predetermined state.

Device Headers

A device header is required at the beginning of a device driver.

FORMAT A device header contains the following bytes in this format:

Offset	Contents
0	Pointer to Next Device
4	Attribute
6	Pointer to Device Strategy Entry Point
7	Pointer to Device Interrupt Entry Point
10	Device Name or Size

Pointer to Next Device

This 2-word (4-byte) entry contains the entry point of the next device header. The first word contains the offset and the second the segment. MS-DOS chains the device headers together using this entry.

If there is one device header in your driver, set this entry to -1. If there is more than one device header, set the first word to the offset and the second to the segment of the next Device Header.

Attribute

This word contains the attributes which tell the system whether this device is a block or character device via Bit 15. The other bits are used to give selected character devices special treatment and are meaningless on a block device. The attribute has the following format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cnts	T Y P E	C T R L	I B M	RESERVED								S P E C	C L K	N U L	O U T	I N

BIT 15 — TYPE

Set the **TYPE** bit to “0” for block devices and “1” for character devices.

Note: For character devices set bits 0 through 4. Set bit 13 for block devices.

BIT 0 — IN

The **IN** bit is set to “1” if the device is the current standard input device.

BIT 1 — OUT

The **OUT** bit is set to “1” if the device is the current standard output device.

BIT 2 — NUL

The **NUL** bit is set to “1” if the device is the current nul device. Set to “0” for all other devices. Although there is a NUL device attribute, it is reserved for MS-DOS.

BIT 3 — CLK

The **CLK** bit is set to "1" if the device is the current clock device. Set to "0" for all other devices.

BIT 4 — SPEC

The **SPEC** bit is set to "1" if the device is the special device. Set to "0" for all other devices.

BITS 5 THROUGH 12 — RESERVED

These bits are reserved for use by MS-DOS. Set all to "0".

BIT 13 — IBM

The **IBM** bit (also referred to as the **NON-IBM FORMAT** bit) is set to "1" if the block device is a **NON-IBM** format. Set to "0" for **IBM** format. The **NON-IBM FORMAT** bit applies only to block devices and affects the operation of the **BUILD BPB** (**BIOS Parameter Block**) device call. The implementation of all block devices is **IBM** software and hardware compatible.

BIT 14 — CTRL

The **CTRL** bit (also referred to as the **IOCTL** bit) is set to "1" when the device can process control strings.

The **IOCTL** bit is meaningful for both types of devices. This bit tells MS-DOS whether the device can handle control strings with the **IOCTL** system call, Function 44H.

If a driver cannot process control strings, this bit is 0. MS-DOS returns an error if an attempt is made to handle control strings. A device which can process control strings sets the IOCTL bit to 1. For drivers of this type, MS-DOS calls IOCTL INPUT and OUTPUT device functions to send and receive IOCTL strings.

The IOCTL functions allow data outside of the normal user's reads and writes to be sent to the driver. The interpretation of this information is up to the driver.

EXAMPLE

For example, assume that you have a new standard input and output device driver. Besides installing the driver, you must tell MS-DOS that you want this new driver to override the current standard input and standard output device.

This is accomplished by setting the attributes to the desired characteristics; so you set Bits 0 and 1 to "1". The attributes entry would look like this:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Data	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	
Entry	80H									03H							

**Pointer to
Device
Strategy
Entry Point**

This word contains the offset of the entry point for the strategy routine. The strategy routine must be located in the same segment as the device header. Strategy routines are called with ES:BX pointing to the Request Header. The interrupt routines get the pointers to the Request Header from the queue that the strategy routines store them in. The command code in the Request Header tells the driver which function to perform.

All 4-byte pointers are stored offset first, then segment.

**Pointer to
Device
Interrupt
Entry Point**

This word contains the offset of the entry point for the interrupt routine. The interrupt routine must be located in the same segment as the device header.

**Device Name
or Size**

These 8 bytes contain the name of a character device or the number of units of a block device. If it is a block device, the number of units can be put in the first byte. This is optional because MS-DOS fills in this location with the value returned by the driver's INIT code.

Request Header

When MS-DOS calls a device driver to perform a function, it passes a Request Header in ES:BX to the strategy entry point. This is a fixed length header followed by data pertinent to the operation being performed. It is the device driver's responsibility to preserve the machine state. For example, save all registers on entry and restore them on exit. There is enough room on the stack when strategy or interrupt is called to do about 20 pushes. If more stack is needed, the driver sets up its own stack.

FORMAT The Request Header contains 13 bytes in this format:

Offset	Contents
0	RECORD LENGTH
1	UNIT CODE
2	COMMAND CODE
3	STATUS
5	RESERVED

RECORD LENGTH This byte in the Request Header contains the length, in bytes, of the pertinent data plus the Request Header.

UNIT CODE This byte contains the unit code of this operation (minor device). This code has no meaning with character devices. If your device driver has three units, then the possible values of the unit code field are 0, 1, and 2.

**COMMAND
CODE**

This byte in the Request Header contains the command code which can have the following:

Code	Function
00	INIT
01	MEDIA CHECK (Block only, NOP for character)
02	BUILD BPB "
03	IOCTL INPUT (Only called if device has IOCTL)
04	INPUT (Read)
05	NON-DESTRUCTIVE INPUT NO WAIT (Char devices only)
06	INPUT STATUS "
07	INPUT FLUSH "
08	OUTPUT (Write)
09	OUTPUT (Write) WITH VERIFY
0AH	OUTPUT STATUS "
0BH	OUTPUT FLUSH "
0CH	IOCTL OUTPUT (Only called if device has IOCTL)
0DH	DEVICE OPEN (Only called if OPEN/CLOSE/RM bit set)
0EH	DEVICE CLOSE "
0FH	REMOVABLE MEDIA (Only called if OPEN/CLOSE/RM bit set and device is block)
10H	OUTPUT UNTIL BUSY (Only called if bit 13 is set on character devices)

Code 0**INIT**

The INIT command code is performed only once; when the device is installed. It returns a location DS:DX which is a pointer to the first free byte of memory after the device driver. To save space, this pointer method can be used to delete initialization code that is only used once.

The INIT routine has the following format:

Offset**Contents**

0	REQUEST HEADER
13	Media Descriptor
14	Break Address
18	Pointer to BPB (not set by character devices)

The number of units, break address, and BPB pointer are set by the driver. On entry, the BPB pointer points to the character after the "=" on the line in CONFIG.SYS. This allows drivers to scan the CONFIG.SYS invocation line for arguments.

If there are multiple device drivers in a single .COM file, the ending address returned by the last INIT is the one MS-DOS uses. All of the device drivers in a single .COM file return the same ending address.

Additional information that block drivers return is:

- The number of units
- A pointer to a BPB
- The media descriptor.

Number of Units

The number of units determines the logical device names. This mapping is determined by the position of the driver in the device list and by the number of units on the device.

BPB Blocks

BPB blocks are used to build an internal MS-DOS data structure for each of the units. The driver passes MS-DOS a pointer to an array of n word BPB pointers, where n is the number of units. If all units are the same, they can share a BPB to save space. This array must be before the free space pointer since MS-DOS builds an internal MS-DOS structure starting at this free byte. The defined sector size must be less than or equal to the maximum sector size defined at INIT time; otherwise, the install fails.

Media Descriptor

The media descriptor byte means nothing to MS-DOS. It is passed to drivers so that they know what parameters MS-DOS is currently using for a drive unit.

Block devices are either dumb or smart. A dumb device defines a unit and an internal MS-DOS structure for each possible media drive combination. For example, unit 0 = drive 0 single side, unit 1 = drive 0 double side. In this case, media descriptor bytes mean nothing.

A smart device allows multiple media per unit. In this case, the BPB table returned by INIT defines space large enough to accommodate the largest possible media supported. Smart drivers use the media descriptor byte to pass information about the media currently in a unit.

Code 1

MEDIA CHECK

MEDIA CHECK is used with block devices (disk drives) only. MS-DOS calls MEDIA CHECK first and passes its current Media Descriptor from the disk boot sector. This command compares both copies of the Media Descriptor then sets the Returned Byte accordingly.

The MEDIA CHECK code has the following format:

Offset	Contents
0	REQUEST HEADER
13	Media Descriptor
14	Returned Byte

Returned Byte

In addition to setting the STATUS word in the Request Header, the driver sets the Returned byte to one of the following:

- Media Not Changed (1) — current BPB and media byte are OK.
- Media Changed (-1) — Current BPB and media are wrong. MS-DOS invalidates buffers for this unit and calls the device driver to build the BPB (Code 2).
- Not Sure (0) — If there are dirty buffers for this unit, MS-DOS assumes the DPB and media byte are OK. If nothing is dirty, MS-DOS assumes the media has changed. It invalidates buffers for the unit and calls the device driver to build the BPB.

- **Error** — If an error occurs, MS-DOS sets the STATUS word in the Request Header for the correct error. That is, if the driver cannot return -1 or 1 because it has a door-lock or other interlock mechanism, the STATUS word will indicate the error.

MEDIA CHECK enhances MS-DOS performance because MS-DOS does not reread the FAT for each directory access.

Code 2

BUILD BPB (BIOS Parameter Block)

BUILD BPB is used with block devices (disk drives) only. MS-DOS performs this command under the following conditions:

- If MEDIA CHECK (Code 1) indicates "Media Changed"
- If MEDIA CHECK (Code 1) indicates "Not Sure" and there are no dirty buffers.

The BUILD BPB code has the following format:

Offset	Contents
0	REQUEST HEADER
13	Media Descriptor
14	Transfer Address
15	Pointer to BPB

Media Descriptor

The last two hexadecimal characters of the FAT entry are called the Media Descriptor.

The media descriptor has the following format:

Bit	Data
0	0 = Single Sided
	1 = Double Sided
1	0 = Not 8 Sector
	1 = 8 Sector
2	0 = Non-removable
	1 = Removable
3	1
4	1
5	1
6	1
7	0 = 80 Tracks
	1 = Not 80 Tracks

For example, assume that you have a disk drive with the following configuration:

- Single Sided
- 8 Sector
- Removable
- 80 Tracks.

The Media Descriptor would be:

Bit	7	6	5	4	3	2	1	0
Data	0	1	1	1	1	1	1	0

Entry 7H | EH

Currently, the media descriptor byte has been defined for a few media types as follows:

Type	Media Descriptor	Capacity
5 ¼ in.	FE	160K
	FCH	180K
	FFH	320K
	FDH	360K
	F9H	1.2M
Hard	F8H	X

Although these media bytes map directly to FAT ID bytes which must be F8-FFH, media bytes can, in general, be any value in the range 00-FFH.

Transfer Address

If Bit 13 (NON IBM FORMAT) in the ATTRIBUTES of the device header is 1, then the transfer address points to a sector scratch buffer.

If Bit 13 (NON IBM FORMAT) is 0, then this buffer contains the first sector of the first FAT and the driver must not alter this buffer.

The first sector of the first FAT must be located in the same sector for all media. This is because the FAT sector is read BEFORE the media is actually determined. Use this mode to read the FAT ID byte.

Pointer to BPB

In addition to setting status word, the driver must set the pointer to the BPB on return.

Codes
3,4,8,9,
C, & 10H

IOCTL INPUT, INPUT, OUTPUT,
OUTPUT WITH VERIFY, IOCTL
OUTPUT

These codes have the following format:

Offset	Contents
0	REQUEST HEADER
13	Media Descriptor
14	Transfer Address
18	Byte/sector Count
20	Starting Sector Number (ignored on character devices)

In addition to setting the status word, the driver must set the sector count to the actual number of sectors (or bytes) transferred. No error check is performed on an IOCTL I/O call. The driver must set the return sector (byte) count to the actual number of bytes transferred.

A user program can not request an I/O of more than FFFFH bytes and cannot wrap around in the transfer segment.

Code 5

NON-DESTRUCTIVE INPUT NO WAIT

The NON-DESTRUCTIVE INPUT NO WAIT code has the following format:

Offset	Contents
0	REQUEST HEADER
13	Read From Device

If the character device returns busy bit = 0 (characters in buffer), then the next character that would be read is returned. This character is not removed from the input buffer, hence the term Non Destructive Read. This call allows MS-DOS to look ahead one input character.

Codes 6 and AH

INPUT STATUS, OUTPUT STATUS

These codes have the following format:

Offset	Contents
0	REQUEST HEADER

This driver sets the status word and the busy bit as follows:

- **For output on character devices without a buffer:**

If bit 9 is 1 on return, a write request waits for completion of a current request. If it is 0, there is no current request and a write request starts immediately.

- **For input on character devices with a buffer:**

A return of 1 means a read request goes to the physical device. If it is 0 on return, then there are characters in the device's buffer and a read returns quickly. A return of 0 also indicates that the user has typed something. MS-DOS assumes that all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer return busy = 0 so that the MS-DOS does not wait for something to get into a non-existent buffer.

**Codes 7 and
BH**

INPUT FLUSH, OUTPUT FLUSH

These codes have the following format:

Offset	Contents
0	REQUEST HEADER

The FLUSH call tells the driver to terminate all pending requests. This call is used to flush the input queue on character devices.

**Codes DH
and EH**

DEVICE OPEN, DEVICE CLOSE

These codes have the following format:

Offset	Contents
0	REQUEST HEADER

These functions are only called by MS-DOS 3.X if the device driver sets the OPEN/CLOSE/RM attribute bit in the device header. They are designed to inform the device about current file activity on the device.

On block devices, they can be used to manage local buffering. The device can keep a reference count. Every OPEN causes the device to increment the count, every CLOSE to decrement. When the count goes to zero; it means there are no open files on the device, and the device should flush any buffers that have been written to that may have been used inside the device because it is now "legal" for the user to change the media on a removable media drive.

There are situations that can occur with this mechanism on block devices because programs that use FCB calls can open files without closing them. It is therefore advisable to reset the count to zero without flushing the buffers when the answer to "has the media been changed?" is yes and the BUILD BPB call is made to the device.

These functions are of more use on character devices. The OPEN function can be used to send a device initialization string. On a printer, this could cause a string for setting font and page size characteristics to be sent to the printer so that it would always be in a known state at the start of an I/O data stream.

Using IOCTL to set these pre- and post-strings provides a flexible mechanism of serial I/O data stream control. The reference count mechanism can also be used to detect a simultaneous access error. It may be desirable to disallow more than one OPEN on a device at any given time. In this case, a second OPEN would result in an error.

Note: All processes have access to stdin, stdout, stderr, stdaux, and stdprn (handles 0,1,2,3,4). Therefore, the CON, AUX, and PRN devices are **always** open.

Code FH

REMOVABLE MEDIA

This code has the following format:

Offset

Contents

0

REQUEST HEADER

This function is only called by MS-DOS 3.X if the device driver sets the OPEN/CLOSE/RM attribute bit in the device header. This function is given only to block devices by a sub-function of the IOCTL system call.

It is sometimes desirable for a utility to know whether it is dealing with a non-removable media drive (such as a fixed disk), or a removable media drive (like a floppy).

An example is the FORMAT utility which prints different versions of some of the prompts.

Removable media information is returned in the busy bit of the status word. If the busy bit is "1" then the media is non-removable. If the busy bit is "0" then the media is removable. Note that no checking of the error bit is performed.

It is assumed that this function always succeeds.

STATUS

This word (2 bytes) in the Request Header contains the status of the command. The STATUS word has the following format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cnts	E R O R	RESERVED					B U S Y	D O N E	ERROR CODE (bit 15 on)							

The status word is set by the driver interrupt routine.

Bit 15**Error (EROR)**

Bit 15 is the error bit. If it is set, then the lower 8 byte (bits 7 - 0) indicate the error. The errors are:

Code	Error
0	Write protect violation
1	Unknown Unit
2	Drive not ready
3	Unknown command
4	CRC error
5	Bad drive request structure length
6	Seek error
7	Unknown media
8	Sector not found
9	Printer out of paper
A	Write Fault
B	Read Fault
C	General failure

Bit 9

Busy

Bit 9 is the busy bit, which is set only by status calls.

- **For output on character devices without a buffer:**

If bit 9 is "1" on return, a write request waits for completion of a current request. If it is "0", there is no current request and a write request starts immediately.

- **For input on character devices with a buffer:**

If bit 9 is "1" on return, a read request goes to the physical device. If it is "0" on return, then there are characters in the device buffer and a read returns quickly. MS-DOS assumes all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer return busy= so that MS-DOS does not wait for non-existent buffer input.

Bit 8

Done

Bit 8 is the done bit. When set, it means the operation is complete.

RESERVED

These eight bits are reserved.

Hardware Controller Documentation

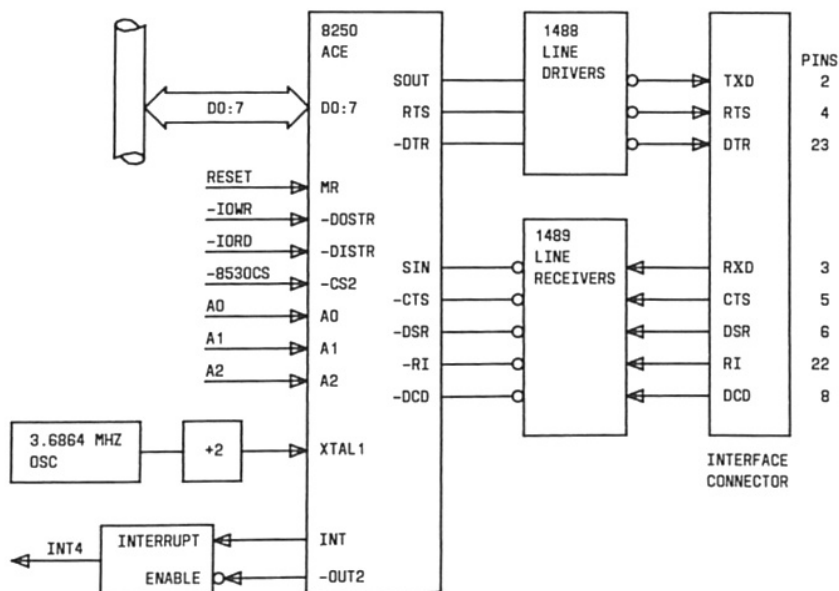
The following parts describe hardware controllers located on the PC 6300 PLUS motherboard. While we describe their internal operations in this guide, you should refer to the manufacturer's programming manuals for complete documentation of each device. The hardware controllers follow in alphabetical order.

Asynchronous Communications Element (ACE)

The Asynchronous Communications Element (ACE) performs serial-to-parallel conversion on input data characters received from a modem and parallel-to-serial conversion on output data characters received from the CPU. You can read the status of transfer operations at any time. This device gives you modem control capability.

The baud rate and serial interface characteristics are programmable. The ACE has a software-tailored interrupt system whose interrupt request is on INT4.

Block Diagram



FUNCTIONS

The Asynchronous Communications Element can perform the following functions:

- Identify interrupts via the Interrupt Identification Register.
- Read the line status via the Line Status Register.
- Read the modem status via the Modem Status Register.
- Receive and send characters via the Data Register.
- Set the Baud rate via the Divisor Latch.
- Set interrupts via the Interrupt Enable Register.
- Write line control characteristics via the Line Control Register.
- Write modem control characteristics via the Modem Control Register.

REGISTERS

PORT		NAME	NOTES
A	B		
3F8	2F8	Data	
3F9	2F9	Interrupt Enable	
3FA	2FA	Interrupt Identification	
3FB	2FB	Line Control	
3FC	2FC	Modem Control	
3FD	2FD	Line Status	
3FE	2FE	Modem Status	
3FF	2FF	Scratch Pad	
3F8	2F8	Divisor Latch	Least Significant byte
3F9	2F9	Divisor Latch	Most Significant byte

REGISTER FORMATS

Interrupt Enable Register (Write Only)

Bit	Contents
7	
6	
5	— N/A
4	
3	— Modem Status Interrupt (Enable=1)
2	— Receiver Line Status Interrupt (Enable=1)
1	— Transmitter Holding Register Empty Interrupt (Enable=1)
0	— Receive Data Ready Interrupt (Enable=1)

Interrupt Identification Register (Read Only)

Bit	Contents
7	
6	
5	— N/A
4	
3	
2	
1	— Interrupt ID
0	— Interrupt Pending? (Yes=0, No=1)

INTERRUPT ID

These two bits indicate the modem status as follows:

- 0 — N/A
- 1 — Transmitter holding register is empty
- 2 — Receiver data is ready
- 3 — Receiver line status

Line Control Register (Read & Write)

Bit	Contents
7	— Access Divisor Latches of the Baud Generator (=1)
6	— Serial output is forced to spacing state (Logic 0). Used to alert terminal in communication system (=1).
5	— Reverse meaning of type of parity (yes=1)
4	— Parity (Even=1, Odd=0)
3	— Parity bit generated for transmit data and checked for in received data (=1)
2	— Stop Bits
1	— Word Length
0	

STOP BITS

This bit indicates the number of stop bits as follows:

- 0 — 1 stop bit
- 1 — 1-1/2 if **Word Length** is 5 bits
- 2 if **Word Length** is 6, 7, or 8 bits

WORD LENGTH

These two bits indicate the word length as follows:

- 0 — 5 bits
- 1 — 6 bits
- 2 — 7 bits
- 3 — 8 bits

Modem Control Register (Read & Write)

Bit	Contents
7	
6	— N/A
5	
4	— Local Loop-Back Feature for diagnostic testing (Set=1)
3	— Forces output 2, an auxiliary user designated output to 0 (=1)
2	— Forces output 1, an auxiliary user designated output to 0 (=1)
1	— Request to Send (=1)
0	— Data Terminal Ready (=1)

Line Status Register (Read Only)

Bit	Contents
7	— Bits 1 - 4 of the register are reset to 0 whenever the register is read. (=0)
6	— Character?
5	— Transmitter Holding Register is empty (Yes=1)
4	— Break Interrupt (Yes=1)
3	— Framing Error (Yes=1)
2	— Parity Error (Yes=1)
1	— Overrun Error (Yes=1)
0	— Data Ready (Yes=1)

CHARACTER?

This bit indicates whether a character is present in the Transmitter Holding or Transmitter Shift register as follows:

- 0 — Yes
- 1 — No

BREAK INTERRUPT

This bit indicates that the received data is in spacing state for full word transmission timer.

FRAMING ERROR

This bit indicates that the received character has an invalid stop bit.

PARITY ERROR

This bit indicates that the received data has incorrect parity.

OVERRUN ERROR

This bit indicates that the character in the receiver buffer was not read before another character arrived.

DATA READY

This bit indicates that the receiver buffer has a complete character.

Modem Status Register (Read & Write)

Bit	Contents
7	— Bits 0 and 1 in the Modem Control register are reset (0) whenever the register is read (Yes=1)
6	— Ring Indicator (Yes=1)
5	— Data Set Ready (Yes=1)
4	— Clear to Send (Yes=1)
3	— Delta Data Carrier Detect (Yes=1)
2	— Trailing Edge of Ring Indicator (Yes=1)
1	— Delta Data Set Ready (Yes=1)
0	— Delta Clear to Send (Yes=1)

DELTA DATA CARRIER DETECT

This bit indicates that the -DCD input has changed status.

TRAILING EDGE OF RING INDICATOR

This bit indicates that the -R1 input has changed from 1 to 0.

DELTA DATA SET READY

This bit indicates that the modem has changed status since the last read.

DELTA CLEAR TO SEND

This bit indicates that the modem has changed status since the last read.

**SEQUENCING
AND
TIMING**

To transmit a character, first issue a Request to Send and Data Terminal Ready to the Modem Control Register. Then wait for the Modem Status to have Data Set Ready and Clear to Send set. When the Transmitter Holding Register is empty as indicated in the Line Status Register, write the character to the data register.

To receive a character, set Data Terminal Ready in the Modem Control Register. Then wait for Data Set Ready in the Modem Status Register. When Data Ready in the Line Status Register is set, input the character from the data register.

**SETTING
THE BAUD
RATE**

To set the baud rate, place the appropriate data in the **Divisor Latch** from the following:

Baud Rate	Data
110	1047
150	768
300	384
600	192
1200	96
1800	64
2000	58
2400	48
3600	32
4800	24
7200	16
9600	12

The data is actually the Divisor which yields the correct rate when applied with the clock frequency in this formula:

$$16 \times \text{DIVISOR} = \text{CLOCK FREQ. (1.8432 MHz)} / \text{BAUD RATE} \times 16$$

Therefore,

$$\text{DIVISOR} = \text{CLOCK FREQ} / \text{BAUD RATE}$$

Example

This program sets the baud rate to 1200 baud:

```

line_ctl equ 3BF
dvsr_l   equ 3F8
dvsr_m   equ 3F9
.
.
.
set_baud: mov al,80H           ;access divisor
          mov dx,line_ctl      ;latch
          out dx,al
          mov al,96H           ;1200 baud divisor
          mov dx,dvsr_l
          out dx,al            ;least significant byte
          mov dx,dvsr_m
          mov al,ah
          out dx,al            ;most significant byte
          xor al,al            ;turn off access to latch
          mov dx,line_ctl
          out dx,al
          ret

```

**BREAK
CONTROL
FEATURE**

To use the break control feature to alert a terminal in a communication system, the following sequence assures that no erroneous or extraneous characters are transmitted.

- Load an all 0's pad character into the transmitter holding register.
- Set break when the transmitter holding register is empty.
- Wait for transmitter empty (Bit 6 = 1 in Line Status Register) and clear break.

Note: The transmitter operates normally during a break sequence and can be used as a character timer to establish an accurate break length.

**INTERRUPT
PRIORITY**

If the ACE is programmed to interrupt, the interrupt is on INT4. The ACE acknowledges the highest priority interrupt as indicated in this chart. The Interrupt Identification Register states which interrupt is pending.

Register	Bit	If Reset (0)
Line Status	Overrun Error (1) Parity Error (2) Framing Error (3) Break Interrupt (4)	Read register
Interrupt Identification	Receiver data ready (1&2)	Read data
Line Status	Transmitter holding Register empty (5)	Write data
Modem Status	Clear to send (4) Data set ready (5) Ring indicator (6) Data carrier detect (3)	Read register

Communications Manager Interface

The Communications Manager is an optional, multiple-speed, intelligent voice/data modem which fits into an 8-bit expansion slot. It supports data rates from 0 to 300 or 1200 bps.

This modem is designed for half or full duplex data transmission applications over the Public Telephone Network. The Communications Manager is compatible with existing 212A-type modems. It may be configured (using hardware straps) for either two-line (providing simultaneous voice/data capabilities) or single-line operation (for alternate voice/data capabilities).

A Communications Manager software package is provided with the hardware. This package supports a 200-entry directory that has simple-to-use editing, calling, redialing, and timing features. The Communications Manager 2.0 version supports the "AT" command set in addition to the Communications Manager software. Also, a Help menu is accessible from almost anywhere within the system.

The Communications Manager hardware consists of a single printed circuit card. The card interfaces with the PC 6300 PLUS through the Bus Expansion Interface which provides data, address, and peripheral control signals. Power for the cards is also supplied through the Bus Expansion Interface.

Note: The "AT" command set is used in modems manufactured by Hayes Microcomputer Products, Inc.

FUNCTIONS

The Communications Manager Interface (modem) provides the following:

- Context switch software allowing the user to switch easily between the communications manager tasks and other tasks.
- Automatically dialing for both voice and data calls from a 200 entry directory.
- Ring detection with auto-answer on a user specified ring count.
- Either touch-tone or pulse dialing.
- Automatic call termination after a user-specified number of ringbacks.
- Hayes compatibility - (Communications Manager 2.0 version)

REGISTERS (NATIONAL 8250)

PORT		DLAB	NAME	NOTES
COM1	COM2	BIT		
3F8	2F8	0	Transmit Buffer	
3F8	2F8	0	Receive Buffer	
3F8	2F8	1	Divisor Latch (LSB)	Set Bit 7 of Line Control Register
3F8	2F8	1	Divisor Latch (MSB)	Set Bit 7 of Line Control Register
3F9	2F9	X	Interrupt Enable	
3FA	2FA	X	Interrupt Identification	
3FB	2FB	X	Line Control	
3FC	2FC	X	Modem Control	
3FD	2FD	X	Line Status	
3FE	2FE	X	Modem Status	

REGISTER FORMATS (NATIONAL 8250)

Interrupt Enable Register

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— Modem Status interrupt
2	— Receive Line Status interrupt
1	— Transmit Holding register interrupt
0	— Data Available interrupt

Interrupt Identification Register

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Interrupt ID Bit 1
2	— Interrupt ID Bit 0
0	— Interrupt pending

Line Control Register

Bit	Contents
7	— Divisor Latch Access bit
6	— Set/Break
5	— Stick Parity
4	— Even Parity select
3	— Parity enabled
2	— Number of stop bits
1	— Word Length select
0	

Modem Control Register

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— Loop
3	— Out 2
2	— Out 1
1	— Request to Send
0	— Data terminal ready

Line Status Register

Bit	Contents
7	— 0
6	— Transmit Shift register empty
5	— Transmit Holding register empty
4	— Break interrupt
3	— Framing error
2	— Parity error
1	— Overrun error
0	— Data ready

Modem Status Register

Bit	Contents
7	— Receive line signal detect
6	— Ring Indicator
5	— Data set ready
4	— Clear to send
3	— Delta Rx Line Signal detect
2	— Trailing Edge Ring indicator
1	— Delta data set ready
0	— Delta clear to send

REGISTERS (INTEL 8251)

PORT		BYTE#	NAME
COM1	COM2		
3FF	2FF	1	Mode
3FF	2FF	2	Command
3FF	2FF	3	Status

REGISTER FORMATS (INTEL 8251)

Mode Register

Bit	Contents
7	— Number Stop Bits
6	
5	— Parity (0=even, 1=odd)
4	— Parity Enable (Yes=1)
3	— Character Length
2	
1	— Baud Rate
0	

Command Register

Bit	Contents
7	— HUNT Mode
6	— Interrupt reset
5	— RTS
4	— Error reset
3	— Send break
2	— Receiver enable (Yes=1)
1	— DTR
0	— Transmitter enable (Yes=1)

Status Register

Bit	Contents
7	— DSR
6	— Sync detect
5	— Framing error
4	— Overrun error
3	— Parity error
2	— Transmitter
1	— Receiver ready
0	— Transmitter ready

COMMENTS

The Communications Manager Interface is accessed by addressing either of the standard communications ports: COM1 or COM2. The desired port is chosen by selecting the proper position for a jumper on the card. The communications port addresses are:

Port	Addresses
COM1	3F8 through 3FFH
COM2	2F8 through 2FFH

Interrupts

The interrupt levels used by the Communication Manager Interface are associated with the COM port selected, as follows:

Port	Level
COM1	INT 4
COM2	INT 3

Bus Interface Devices

There are two devices which have access to the bus on the Communications Manager Interface: a National 8250 UART (Universal Asynchronous Receiver/Transmitter) and an Intel 8251 UART. Using two UART's permits two logical channels: data (National 8250) and commands (Intel 8251).

The Communications Manager Interface supports both the Communications Manager (see COMMAND PROTOCOL) and "AT" (Hayes) command sets. The "AT" protocol does not have a separate channel for commands. That is, both data and commands are sent via the National 8250 UART.

Which mode (AT or CM) the Communications Manager Interface uses depends on the software. At system reset, the interface is set for AT mode (single channel). Communications Manager software will automatically set the interface for dual channel.

The channels are accessed as follows in CM mode:

Channel	Addresses
DATA	3F8 through 3FEH (COM1)
DATA	2F8 through 2FEH (COM2)
COMMAND	3FFH (COM1)
COMMAND	2FFH (COM2)

In the AT mode, both command and data use a single channel. The channel is accessed at address:

3F8 through 3FEH
or
2F8 through 2FEH

Also, the AT command set is used.

NATIONAL 8250 UART MODE CONTROL

The National 8250 UART modes are selected by writing to it's internal registers. See

REGISTERS (NATIONAL 8250) for a complete breakdown of registers.

INTEL 8251 UART MODE CONTROL

The Intel 8251 UART registers are **not** directly addressable. They are accessed by a sequential state machine. That is, commands and data must be sent in a proper order.

A control lead (C/D) is used to select between command and data. In the Communications Manager Interface, the C/D lead is controlled by Bit 2 (OUT 1) of the Modem Control register (National 8250). The OUT 1 pin of the National 8250 is hardwired to the C/D lead of the Intel 8251. Update this bit whenever the command/data mode of the Intel 8251 is changed.

A typical block of data for the Intel 8251 would look like this:

Contents	C/D Lead
Hardware Reset	N/A
Mode Instruction	1
Command Instruction	1
Data (read/write)	0
Data	0
Data	0
.	
.	
.	
Status (read)	1
Command Instruction (write)	1
Data	0
Data	0
.	
.	
.	

National 8250 UART Pin Assignments

The National 8250 UART has the following pin assignments:

Pin(s)	Label	Notes
1-8	DATA	
9	RCLOCK	Tied to BAUDOUT (Pin 15)
10	SIN	Transmitted data (serial)
11	SOUT	Received data (serial)
12	CS0	Chip Select (tied active)
13	CS1	Chip Select (active high)
14	CS2	Chip Select (active low)
15	BAUDOUT	Transmit clock (Pin 9) (active low)
16	XTAL1	Crystal oscillator (input for 1.8432 Mhz)
17	XTAL2	Crystal oscillator (input for 1.8432 Mhz)
18	WR	Connected to bus IOWRITE (active low)
19	DOSTR	Data Out strobe (tied active)
20	GROUND	
21	DISTR	Data In strobe (tied to BUS IOREAD) (active low)
22	DISTR	Data In strobe (tied inactive)
23	DDIS	Driver Disable (no connect)
24	CSOUT	Chip Select out (no connect)
25	ADS	Address strobe (no connect) (active low)
26	A2	BUS address A2
27	A1	BUS address A1
28	A0	BUS address A0
29		N/A
30	INTR	Interrupt output (drives INT3 or 4)
31	OUT2	Disables all interrupts (active low)
32	RTS	Request send (tied to CTS) (active low)
33	DTR	Data terminal ready (active low)
34	OUT1	Connected to C/D lead of Intel 8251 (active low)
35	MR	RESET (tied to system reset)
36	CTS	Clear to send (tied to RTS) (active low)
37	DSR	Data set ready (tied active) (active low)
38	DCD	Data carrier detect (active low)
39	RI	Ring indicator (driven by firmware) (active low)
40	+5 volts	

**Intel 8251
UART Pin
Assignments**

The Intel 8251 UART has the following pin assignments:

Pin(s)	Label	Notes
1-2	DATA	
3	RXD	Received data (serial)
4	GROUND	
5-8	DATA	
9	TXC	Transmit clock (19.2 kHz) (active low)
10	WR	Write (tied to system IOWRITE) (active low)
11	CS	Chip Select (active low)
12	C/D	Command/Data select lead
13	RD	Read (tied to system IOREAD) (active low)
14	RXRDY	Receiver ready (OR'd with 8250 interrupt out)
15	TXRDY	No connection
16	SYNDET	No connection
17	CTS	Clear to send (tied to RTS) (active low)
18	TXE	No connection
19	TXD	Transmitted data (serial)
20	CLK	Clock (2.45 Mhz)
21	RST	Tied to system reset
22	DSR	Tied active low
23	RTS	Request to send (tied to CTS) (active low)
24	DTR	Data terminal ready (0=AT mode,1=CM mode)
25	RXC	Receiver clock (19.2 Mhz)
26	+5 volts	

**COMMAND
PROTOCOL**

The following is a detailed description of the Command/Response Protocol used in the CM mode.

The terms Global and Local are used to define the effect of setting two different switches, both of which are used in the auto-dialing process. Setting a switch Globally will effect all numbers auto-dialed from that point on. If a particular telephone number requires different switch settings, they may be embedded in the telephone number designation for Local use and do not affect the Global settings.

> — ready prompt.

This response is given after a power-up reset (10-second delay) and when there is no activity on either line.

R — unconditional reset command.

All current processes are terminated and Communications Manager hardware is reconfigured. Variables take on their default values.

L — line configuration and status.

You can use this command at any time to determine the number of lines the Communications Manager is configured for, the current state of the receiver hook, and whether or not the modem is in the data mode. The responses to this command are, in order:

S (single line) or **M** (multi-line)

H (currently off-hook) or **h** (currently on-hook)

D (data up) or **d** (not in data mode).

The firmware version number (ASCII 0 through 9) is the last character received.

T — select touch-tone dialing Global and Local.

This command, in its global sense, causes all auto-dialed numbers to be touch-tone dialing as opposed to pulse dialing. This command may also be locally embedded anywhere within a telephone number designation to force touch-tone dialing of all digits following the **T**. This local use does not affect the global switch.

The response to this command is an echoed **T**.

P — select pulse dialing Global and Local.

This command is identical to the **T** command except that the response is an echoed **P**.

C — call-progress-tone (CPT) detection switch ON Global and Local.

You may use this command at any time to turn on CPT detection globally. All auto-dialed calls following this command will have call progress monitored by the Communications Manager with appropriate responses. You may also use this command locally within a phone number designation if CPT detection is necessary when the global switch is OFF (see **c** command). The local use of this command does not affect the global switch. The response to this command is an echoed **C**. It should be noted that at the end of voice auto-dialing when CPT detection is ON, if the user goes manually off-hook, CPT detection ceases and control is turned over to the user.

c — call-progress-tone (CPT) detection switch OFF Global and Local.

This command is given under the same conditions as **C**, except the CPT detection switch is turned OFF. The response is an echoed **c**.

b[s,n] — baud rate select.

You may use this command at any time to select slow (300), or normal (1200) baud rates. The next time a data call is originated, the selected baud rate will be used. The response to this command is an echoed **b** followed by either **s**, or **n** as given.

a[n,CR] — set auto-answer count or answer immediately.

Use this command to either set the auto-answer count (**an**) or to answer immediately (**a(CR)**) after ringing is detected. The auto-answer count command may be given at any time. The value “n” (0 - 255) may be up to three ASCII numerics (0-9). If less than three characters are needed, a CR must be given to terminate the count designation.

For example, **a6(CR)** is for auto-answer on the sixth ring. The default value is zero, meaning auto-answering will never occur. The response to this command is the echoing of each character as given. Once ringing has been detected, each ring cycle will be counted. When the number of ring cycles equals the auto-answer count, the Communications Manager will auto-answer and respond with an **A**. If ringing has been detected (on line 1, or line 2 for 2-line configuration), the answer immediate command **a(CR)** may be given and an attempt to go into the data mode will be made.

The response to this command is an echoed **aA**. Note that the **(CR)** is replaced by **A**. This is to be consistent with the auto-answering response.

tn — set terminate on “n” ringbacks.

You may use this command at any time to set the variable that will terminate all calls after “n” ringbacks have been detected. The CPT detection switch must be ON at the time. The designation for “n” is the same as above (**an**). The response to this command is the echoing of each character as given. The default value is 6.

v or **d**[**T,P,C,c,f,COMMA,0-9,#,***](**CR**) — voice and data call originate.

Use this command string to originate voice (**v**[]) and data (**d**[]) calls. Note the use of local switches **T,P,C**, and **c**.

The **f** is used to indicate switchhook **flash**. **COMMA** is used wherever secondary dial-tone is to be detected before proceeding. If the CPT detection switch is OFF at this time, a 2.5 second delay will occur before proceeding. The response to this command string is the echoing of each character as given.

Once the Communications Manager receives the (**CR**), it will begin auto-dialing. Assuming all dial-tones are detected, either a **g** (voice call) or a **G** (data call) will be returned to indicate the end of dialing. The length of the telephone number (excluding the **v** or **d** but including local switches) may be up to 50 characters.

The Communications Manager will store this number for redial if necessary. This can be accomplished by the command **v(CR)** or **d(CR)**. The Communications Manager will also store both a voice and data number for redial as long as the combined length is not more than 50 characters. If the combined length is greater than 50 characters, only the last number given is available for redial (the first number was over-written by the last number).

For example, if a 24 character voice number is given followed by a 30 character data number, any attempt to redial the voice number will result in dialing fewer than the expected number of digits. It should be noted that subsequent changes of global switches do not change the local switches in a stored telephone number designation.

^v,^d — terminate current voice or data activity.

Use this command to terminate auto-dialing, CPT detection, or a data transfer.

x — disconnect.

If the user is manually off-hook, giving this command will immediately disconnect the line and return dial-tone. The response is an echoed **x**.

f — switchhook flash.

Use this command whenever the user is manually off-hook, or may be embedded in a telephone number designation as needed. The response is the echoed **f**.

m — voice <--> data mode switching.

This command enables you to switch from voice to data mode and back to voice, indefinitely, without dropping the line. Mode switching can only occur on line 1 if line 2 (2-line configuration) is inactive. This means the modem is available to be switched onto line 1. You can only use this command when the user is manually off-hook. The response acknowledgment is the echoed command **m**. It may be thought of as a "toggle" command.

H — manually off-hook.

This response occurs whenever the user manually goes off-hook.

h — manually on-hook.

This response occurs whenever the user manually goes back on-hook. This response will also be returned for on-hook voice calling if, upon the completion of dialing (CPT switch is OFF), the user is still on-hook.

D — data mode entered.

This response will be returned whenever the modem enters the data mode. Either an **s**(300) or an **n**(1200) will also be returned depending on the baud rate detected. This will occur for both answer and originate.

d — loss of data mode or timed-out without entering data mode.

This response will be returned whenever the modem leaves the data mode. Also, if an attempt to enter the data mode fails and an internal 20 second time-out occurs, this response will be given.

i, I — no dial-tone detected in 4.5 seconds.

o, O — receiver off-hook.

Either the local phone set or another line appearance is off-hook and active such that initial dial tone is not detected.

z, Z — busy tone detected.

k, K — check number; no call-progress-tones detected after number has been dialed.

w, W — wait, ringback detected.

Ringback cycles will be counted.

R — response when ringing is detected.

e — end of ringing.

Response when ringing is no longer detected.

COMMAND	DESCRIPTION	RESPONSE
Pwr-up, R	Power-up, reset. Re-initialize hardware.	>
L	Line configuration, status, firmware Version.	SHD Ver Mhd
T, P	Touch-tone or pulse dialing. Global and Local. Default is T.	T, P
C, c	CPT Detection ON,OFF. Global and Local. Default is c.	C, c
an	Auto-answer count. n = 0-255 (ASCII). Default is a0.	an
a(CR)	Answer immediately.	a(CR)
tn	Terminate calls after n ringbacks. Default is t6.	tn
b[sn]	Baud rate select: s(300), n(1200). Default is 1200.	b[sn]
<div style="border: 1px solid black; padding: 2px; display: inline-block;"> [COMMA, v 0-9,*, d #,T,P, C,c,f] </div>	Voice or data call originate.	<div style="border: 1px solid black; padding: 2px; display: inline-block;"> [COMMA, v 0-9,*, d #,T,P, C,c,f] </div>
^v	Terminate current voice activity.	^v
^d	Terminate current data activity.	^d
m	Voice <--> data mode switching.	m
x	Disconnect.	x
f	Switchhook flash.	f

EVENT	DESCRIPTION	RESPONSE
On -> off hook	Manually off-hook	H
Off -> on hook	Manually on-hook	h
Enter DM	Entering data mode. 300 1200	Ds Dn d
Loss of DM or time-out	Data mode terminated or timed-out without entering data mode.	
End of dialing	End of voice or data call dialing	g(voice) G(data)
Ringing	Ringing detected.	R
End of ringing	Ringing stopped.	e
Auto-answering	Auto-answering (immediately or by reaching auto-answer ring count)	A
Ringback	Ringback detected.	w(voice) W(data)
Busy	Busy detected.	z(voice) Z(data)
No dial-tone	No dial-tone detected in 4.5 seconds.	i(voice) I(data)
ROH	Receiver off-hook. Line activity detected instead of dial-tone.	o(voice) O(data)
chknumber	Either no CPT detected or indeterminable.	k(voice) K(data)

Communications Manager Session Example

EVENT/ COMMAND	DESCRIPTION	RESPONSE
HW Reset	System power-up Default conditions set: Touch-tone CPT detect OFF Auto-answer OFF (a0) Term count = 6 Baud rate = 1200	10 second delay >
L	Line configuration command given. Two-lines, manually on- hook, not in data mode and Version 1 firmware.	Mhd1
a1(CR) bs C	Set auto-answer count to first ring. Set baud rate to 300. Set CPT detection ON.	a1(CR) bs C
d9,5551212(CR)	Data call originate at 300 baud.	d9,555 1212(CR)
vP2128c(CR)	Voice call originate using pulse dialing. Initial dial tone will be looked for, but CPT detection is switched OFF after dialing.	vP2128c (CR)
end of data call dialing	Communication Manager has dialed out data call.	G
end of voice call dialing	Voice call has been dialed and because CPT detection is OFF(locally) the h response informs the user to go off-hook.	gh
Ringback detected	Ringback is detected on the data call.	W
User goes manually off-hook	User responds to "end-of-dialing voice" response (gh) and goes off-hook.	H

Communications Manager Session Example

EVENT/ COMMAND	DESCRIPTION	RESPONSE
Data mode up.	Data call completed. Data mode up in 300.	Ds
^d	Data transfer on Line 2 is complete and the user gives the "terminate data" command.	^d
a0(CR)m	User's voice call is still active, but now wants to mode switch to exchange data on the same call. Auto-answer count set to never answer calls on Line 2 while mode switching.	a0(CR)m
Data Mode up.	Mode switching accomplished and up in data mode. Data transfer begins.	Ds
Ring detected.	Incoming call on Line 2.(Will not auto-answer.)	R
m	Recognizing that Line 2 is ringing, the user mode switches back to voice on Line 1 to inform the called party that they must continue later.	m
User goes manually on-hook	User returns on-hook terminating the voice call. Line 2 is still ringing. (e not received)	h
a(CR)	User wants to auto-answer IMMEDIATELY the incoming data call on Line 2.	aA

Communications Manager Session Example

EVENT/ COMMAND	DESCRIPTION	RESPONSE
20-sec time-out	20 seconds elapse without going into data mode. Calling party probably hung-up. Both lines are inactive now and the prompt is returned indicating the IDLE state.	d>
R	Frustrated about missing the incoming call, the user resets Communications Manager and goes for coffee!	no delay >

Display Controller

The PC 6300 PLUS Display Controller interfaces the CPU to either monochrome or color displays. It uses a HD6845 CRT Controller. The Display Controller operates in two basic modes — text or all points addressable (APA) graphics. Several resolutions are available depending on the display mode.

RESOLUTION	PC COMPATIBLE	TEXT/ GRAPHICS	COLOR/ MONOCHROME
80X25	Yes	T	C/M
40X25	Yes	T	C/M
80X26	No	T	C/M
80X27	No	T	C/M
640X400	No	G	M
640X200	Yes	G	M
320X200	Yes	G	C/M

In text mode, character attributes include reverse video, blinking, highlight, hide, and underline. In color mode if blinking is not requested, one of 16 colors can be chosen. Otherwise, one of 8 colors can be chosen.

In graphic mode, each pixel on a color display is one of four selected colors. These four colors are from a choice of 16. In a monochrome display, these 16 colors are shades of green from black (very dark green) to white (very light green).

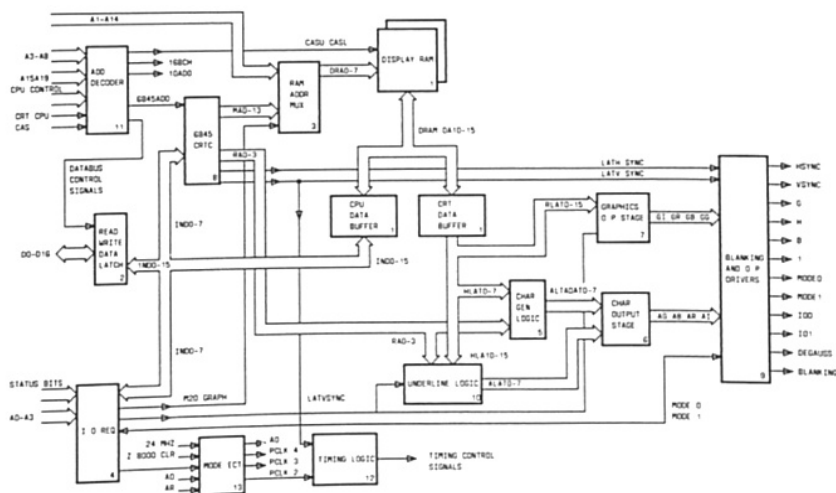
The Display Controller has 32K bytes of RAM to refresh one screen page.

The Display Controller can be upgraded with an optional display enhancement board (DEB).

This board gives you the following features:

- Up to three additional screen pages in RAM
- Software controlled look-up table for character set
- High resolution characters (16 X 16 pixels instead of 8 X 16 pixels)
- Hardware smooth scroll
- The ability to mix text and graphic modes simultaneously
- Up to 16 colors can be displayed simultaneously.

Block Diagram



FUNCTIONS

- Initialize the 16 registers of the HD6845 with predetermined values.
- Set different mode characteristics such as text or graphics, type of graphics, blinking character, etc., via the Mode Select registers.
- Choose the different color or shades of green to display via the Color Select register.
- Set starting and ending line for cursor.
- Set cursor to location in memory.
- Read the current position of the cursor.
- Set the address of the current page to display.

REGISTERS

PORT	NAME
3D8	Mode Select 1
3D9	Color Select
3DA	Status
3DB	Light Pen Reset
3DC	Light Pen Latch Set
3DE	Mode Select 2
3D4	Pointer to HD6845
3D5	HD6845 Data

HD6845 REGISTERS

#	NAME	INITIALIZATION VALUE				
		40X25	80X25	80X26	80X27	GRAPHICS
0	Horizontal Total	38	71	71	71	38
1	Horizontal Displayed	28	50	50	50	28
2	Horizontal SYNC Position	2D	5A	5A	5A	2D
3	Horizontal SYNC Width	06	0C	0C	0c	06
4	Vertical Total	1F	1F	9B	9B	7F
5	Vertical Total Adjust	06	06	8C	8C	06
6	Vertical Displayed	19	19	1A	1B	64
7	Vertical SYNC Position	1C	1C	9B	9B	70
8	Interlace Mode	02	02	02	02	02
9	MAX. Scan Line Address	07	07	8E	8E	01
AH	Cursor Start Line (Size)	06	06	06	06	06
BH	Cursor End Line	07	07	07	07	07
CH	Active Page Start Adrs (H)	00	00	00	00	00
DH	Active Page Start Adrs (L)	00	00	00	00	00
EH	Cursor Address (H)	00	00	00	00	00
FH	Cursor Address (L)	00	00	00	00	00
10H	Light Pen (H)	00	00	00	00	00
11H	Light Pen (L)	00	00	00	00	00

REGISTER FORMATS

Mode Select 1 Register (Write Only)

Bit	Contents
7	—
6	— N/A
5	— Text mode
4	— Graphics resolution
3	— Display
2	— N/A
1	— Mode
0	— Select Text Mode

TEXT MODE

This bit indicates whether blinking or intensified color as follows:

- 0 — Attribute character bit 7 defines background intensity
- 1 — Blinking character

GRAPHICS RESOLUTION

This bit indicates whether the resolution is medium or high as follows:

- 0 — Medium resolution
- 1 — High resolution

DISPLAY

This bit indicates whether the display is enabled or blank as follows:

- 0 — Blank display
- 1 — Enable display

MODE

This bit indicates the mode as follows:

- 0 — Text mode
- 1 — Graphics mode

SELECT TEXT MODE

This bit indicates the selected text mode as follows:

- 0 — 40X25 Text mode
- 1 — 80X25, 80X26, or 80X27 Text mode

Mode Select 2 Register (Write Only)

Bit	Contents
7	— Enable clock
6	— Underline color text (Yes=1)
5	— N/A
4	— N/A
3	— Enable RAM
2	— Select character set
1	— Request degauss
0	— 512 X 256 Graphics (Yes=1)

ENABLE CLOCK

This bit indicates which clock is enabled as follows:

- 0 — 24-MHz clock
- 1 — 19-MHz clock

ENABLE RAM

This bit indicates how much RAM is enabled as follows:

- 0 — 16K byte RAM
- 1 — 32K byte RAM

SELECT CHARACTER SET

This bit indicates which character set is enabled as follows:

- 0 — Primary character set
- 1 — Alternate character set

Color Select Register — Graphics Mode Only (Write Only)

Bit	Contents
7	— N/A
6	
5	— Select Foreground Palette
4	— Set Foreground Intensity
3	— Modify Color Selected? (Yes=1)
2	— Color Selected
1	
0	

SELECT FOREGROUND PALETTE

This bit indicates which foreground palette is selected as follows:

- 0 — Palette 0 (cyan, magenta, and white)
- 1 — Palette 1 (Green, red, and yellow)

FOREGROUND INTENSITY

This bit indicates whether the foreground intensity is set as follows:

- 0 — Off
- 1 — On

COLOR SELECTED

These bits indicate which shade color is selected as follows:

- 0 — Black
- 1 — Blue
- 2 — Green
- 3 — Cyan
- 4 — Red
- 5 — Magenta
- 6 — Brown
- 7 — Light gray

If the **Modify Color Selected** bit is 1, the color is as follows:

- 0 — Dark gray
- 1 — Light blue
- 2 — Light green
- 3 — Light cyan
- 4 — Light red
- 5 — Light magenta
- 6 — Yellow
- 7 — White

Status Register (Read Only)

Bit	Contents
7	— Display option board present? (Yes=0, No=3)
6	
5	— Display
4	
3	— First half of vertical retrace
2	— Light pen switched off? (Yes=1)
1	— Light pen triggered? (Yes=1)
0	— 1 (Horizontal Retracing)

DISPLAY

This bit indicates the type of display as follows:

- 1 — Color display
- 3 — Monochrome display

MODES

Text Mode Every character position is defined as follows:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cnts	BACK				FORE				CHARACTER							

Each character position requires two bytes of memory. One byte is the character code and the other is the attributes of the character. The character has foreground (FORE) and background (BACK) color attributes.

If neither the underline or blinking capabilities are specified, the color choices are:

Data	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	Light Gray
8	Dark Gray
9	Light Blue
A	Light Green
B	Light Cyan
C	Light Red
D	Light Magenta
E	Yellow
F	White

On a monochrome display, the colors are represented in shades of green from black to green as follows:

Data	Color
0	Black
1	Darkest Green
.	.
.	.
.	.
F	Lightest Green

Typical codes for monochrome displays are:

Code	Display
0F	Normal (Light green letters on black)
F0	Reverse video (Black letters on light green)
00	Position black
FF	Position light green

COMMENTS

The first position in the left-hand corner of the screen is defined in the first two bytes of memory starting at B0000H. The next position, one column to the right, is defined in the next two bytes of memory at B0002H, and so on. The first character in the next row follows immediately after the definition for the last character in the first row.

For 80-column X 25-row display, memory looks like this:

Memory	Col 1	Col 2	...	Col 79	Col 80	Memory
B0000	Row 1					B009F
B00A0	Row 2					B013F
B0140	Row 3					B01FF
.	.					
.	.					
.	.					
B0E60	Row 24					B0EFF
B0F00	Row 25					B0F9F

Therefore, the 25 rows of an 80 X 25 display are saved in memory as follows:

Row	Memory Addresses		
	Col. 1	through	Col. 80
1	B0000	-	B009F
2	B00A0	-	B013F
3	B0140	-	B01FF
4	B01E0	-	B027F
5	B0280	-	B031F
6	B0320	-	B03BF
7	B03C0	-	B045F
8	B0460	-	B04FF
9	B0500	-	B059F
10	B05A0	-	B063F
11	B0640	-	B06DF
12	B06E0	-	B077F
13	B0780	-	B081F
14	B0820	-	B08BF
15	B08C0	-	B095F
16	B0960	-	B09FF
17	B0A00	-	B0A9F
18	B0AA0	-	B0B3F
19	B0B40	-	B0BDF
20	B0BE0	-	B0C7F
21	B0C80	-	B0D1F
22	B0D20	-	B0DBF
23	B0DC0	-	B0E5F
24	B0E60	-	B0EFF
25	B0F00	-	B0F9F

The 80-column display uses 4K bytes of RAM and the 40-column display uses 2K bytes of RAM. The rest of the 32K bytes are used for multiple screen images called pages. There are 16 pages available with 40-column displays and 8 available with 80-column displays.

When the blinking capability is specified in Mode Select 1, then Bit 15 of the attribute byte specifies whether the character blinks.

If the underline capability is specified in the Mode Select register 2, then Bit 11 specifies whether the character is underlined.

Graphics Mode

In graphics mode, the display screen is a grid of pixels, the smallest displayable block on a screen.

In medium resolution, there are 640 pixels (columns) per row and either 200 or 400 rows. Each pixel requires 2 bits of memory. Each byte contains 4 pixels in this format:

Bit	7	6	5	4	3	2	1	0
Data	P1		P2		P3		P4	

The actual colors are defined in the Color Select register. That is, bits 0 through 3 determine the background color and bit 5 determines which palette contains the foreground colors.

When bit 5 of the Color Select register is 0, the pixel values (P1 through 4) are:

- 0 — Background color
- 1 — Cyan
- 2 — Magenta
- 3 — White

When bit 5 of the Color Select register is 1, the pixel values are:

- 0 — Background color
- 1 — Green
- 2 — Red
- 3 — Yellow

In high resolution, there are 640 pixels (columns) per row and either 200 or 400 rows.

Each pixel requires 1 bit of memory. Each byte contains 8 pixels in this format:

Bit	7	6	5	4	3	2	1	0
Data	P1	P2	P3	P4	P5	P6	P7	P8

The background color is always black when the pixel is off (0). When the pixel is on (1), the foreground color is defined by bits 0 through 3 in the Color Select register.

Unlike text mode, rows of pixels do not follow one another in memory.

In the 640 X 400 pixel resolution, memory is as follows:

Memory	Row
B8000	0, 4, 8, ..., 396 (8K bytes)
B9F3F	N/A
BA000	1, 5, 9, ..., 397 (8K bytes)
BBF3F	N/A
BC000	2, 6, 10, ..., 398 (8K bytes)
BDF3F	N/A
BE000	3, 7, 11, ..., 399 (8K bytes)
BFF3F	

In the 320 X 200 pixel resolution, memory is as follows:

Memory	Row
B8000	Even (0, 2, 4, ..., 198) Page 0 (8K bytes)
B9F3F	N/A
BA000	Odd (1, 3, 5, ..., 199) Page 0 (8K bytes)
BBF3F	N/A
BC000	Even (0, 2, 4, ..., 198) Page 1 (8K bytes)
BDF3F	N/A
BE000	Odd (1, 3, 5, ..., 199) Page 1 (8K bytes)
BFF3F	

SEQUENCING AND TIMING

There are two methods to communicate with the display. One is with I/O commands and the other is memory mapping. I/O commands are used to set the modes of operation, the cursor position, the cursor size, or the current active page.

The second method is memory mapping. With memory mapping, the CPU reads and writes from the controller memory. The screen is continually scanning this memory and will show the updates.

In text mode, updates must occur during horizontal or vertical retracing. First, calculate the address in memory of the data to access. Then wait until bit 0 or bit 3 in the Status register is set before making the access.

Whenever you change the mode of operation, disable the video by setting Bit 3 in Mode Select register 1 to 0.

To clear the distortion in a color display, especially after it has been moved, the screen must be degaussed as follows:

- Disable the display
- Produce a pulse of between 1 and 10 msec with the Degauss bit (bit 1 in the Mode Select 2 register)
- Enable the display after 5 seconds.

Example

This program changes the display from 80 X 25 text mode to 40 X 25 text mode:

```

model          equ 3d8
change_mode:   mov dx,model      ;adrs of mode select
                                   ;register 1
               mov al,01h        ;80x25
               out dx,al         ;blank display
               mov al,08h        ;enable display
               out dx,al
               ret

```

DMA Controller

The DMA controller (INTEL 8237A) allows devices to transfer data directly to and from memory without CPU involvement. It has four channels:

- Channel 0 is used to refresh memory and has the highest priority. The Interval Timer is programmed to periodically request a dummy DMA transfer. This creates a memory read cycle which refreshes memory.
- Channel 1 supports high-speed transfer between I/O devices and memory. It is available on the I/O expansion bus.
- Channel 2 is dedicated to the floppy disk controller.
- Channel 3 is dedicated to the hard disk unit controller and has the lowest priority.

The DMA controller has four transfer modes:

- The single transfer mode makes only one transfer.
- The block transfer mode continues transferring until the count goes from 0 to FFFFH.
- The demand transfer mode allows transfers to continue until either the count underflows from 0 to FFFH, the process ends, or the DREQ goes inactive.
- The cascade mode allows more than one DMA controller to be used.

When autoinitialize is requested, the original values of the Current Address and Current Count registers are restored at the end of the operation.

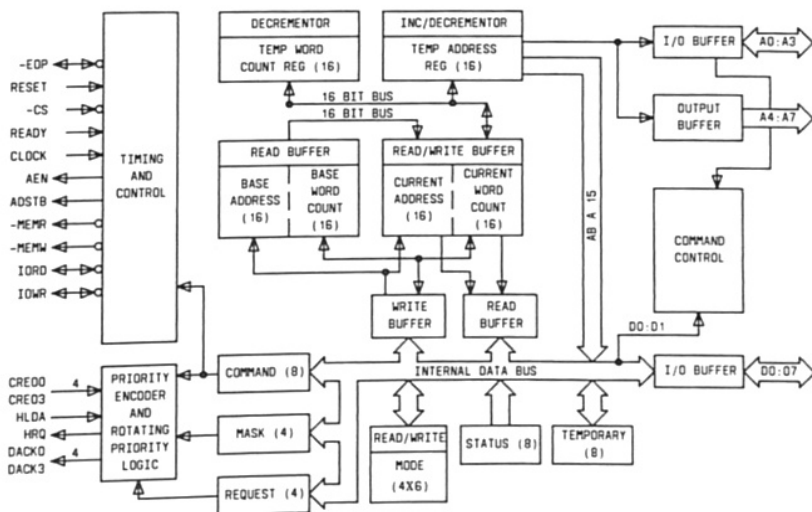
Note: The DMA Accelerator circuitry on the PC 6300 PLUS is designed to operate on DMA channel 3 in "demand" mode. To use channel 3 in a different mode, disable the DMA accelerator with a jumper on the daughter board or by writing "1" to port 63H. The DMA accelerator is re-enabled by writing "0" to port 63H.

The DMA controller has two types of priority schemes.

- The fixed scheme bases the priority on the descending value of their numbers. In this scheme, Channel 3 has the lowest priority.
- The second scheme is rotating priority. That is, the last channel to get service becomes the lowest priority channel.

Compressed Timing allows greater throughput by compressing the transfer time into two clock cycles.

Block Diagram



FUNCTIONS

The DMA Controller can perform the following functions:

- Disable the controller via bit 2 of the Command Register.
- Clear the controller and the Command, Status, Request, Temporary and Flip/Flop Registers.
- Set up the controller for the desired operation via the Mode Register.

SEGMENT NIBBLE

One nibble. The significant nibble of the segment register is in bits 12-15. Rotate to Bits 0-3 before output.

COUNT REGISTER

16-bit 1's complement of the # of bytes. First output the LSB and then the MSB.

- Request for DMA services via the Request Register.
- Read the channel status via the Status Register.
- Control the operation of the DMA controller via the Command Register.

- Enable and disable the automatic DMA transfer for all channels via the Write All Masks Register.
- Enable and Disable the automatic DMA transfers for a single channel via the Mask Register.
- Specify the mode for a specific channel via the Mode Register.
- CLEAR MASKS
Clear all the masks so that all channels accept DMA commands via the Clear Mask register.

REGISTERS

PORT	NAME	NOTES
0000	Channel 0 address	16-bits wide (Write only)
0001	Channel 0 count	integer of # bytes to transfer
0000	Channel 0 current address	16-bits wide (Read only)
0001	Channel 0 current count	integer of # bytes to transfer
0002	Channel 1 address	16-bits wide (Write only)
0003	Channel 1 count	integer of # bytes to transfer
0002	Channel 1 current address	16-bits wide (Read only)
0003	Channel 1 current count	integer of # bytes to transfer
0004	Channel 2 address	16-bits wide (Write only)
0005	Channel 2 count	integer of # bytes to transfer
0004	Channel 2 current address	16-bits wide (Read only)
0005	Channel 2 current count	integer of # bytes to transfer
0006	Channel 3 address	16-bits wide (Write only)
0007	Channel 3 count	integer of # bytes to transfer
0006	Channel 3 current address	16-bits wide (Read only)
0007	Channel 3 current count	integer of # bytes to transfer
0008	Status	
0008	Command	
0009	Request	
000A	Single mask	
000B	Mode	
000C	Clear flip/flop	Execute prior to read or write (Write only)
000D	Temporary	Contains last byte (Read only)
000D	Master clear	Any write clears controller (Write only)
000E	Clear mask	All channels accept DMA commands (Write only)
000F	Write all mask	
0080	Channel 0 segment	8 bits wide (Write only)
0082	Channel 1 segment	8 bits wide (Write only)
0081	Channel 2 segment	8 bits wide (Write only)
0083	Channel 3 segment	8 bits wide (Write only)

REGISTER FORMATS

Status Register (Read Only)

Bit	Contents
7	— Channel 3 has request (Yes=1)
6	— Channel 2 has request (Yes=1)
5	— Channel 1 has request (Yes=1)
4	— Channel 0 has request (Yes=1)
3	— Channel 3 finished transfer (Yes=1)
2	— Channel 2 finished transfer (Yes=1)
1	— Channel 1 finished transfer (Yes=1)
0	— Channel 0 finished transfer (Yes=1)

Command Register (Write Only)

Bit	Contents
7	— DACK level (0=Low, 1=High)
6	— DREQ level (0=High, 1=Low)
5	— Write Selection
4	— Priority
3	— Timing
2	— Controller (Enable=0)
1	— N/A
0	— Memory-to-memory (Enable=1)

WRITE SELECTION

This bit indicates the write selection as follows:

- 0 — Late write selection
- 1 — Extended write selection

PRIORITY

This bit indicates the priority as follows:

- 0 — Fixed
- 1 — Rotating

TIMING

This bit indicates the timing as follows:

- 0 — Normal
- 1 — Compressed

Request Register (Write Only)

Bit	Contents
7	— N/A
6	
5	
4	
3	— Request bit (Reset=0, Set=1)
2	
1	— Select Channel
0	

SELECT CHANNEL

These two bytes indicate the channel select as follows:

- 0 — Channel 0
- 1 — Channel 1
- 2 — Channel 2
- 3 — Channel 3

Single Mask Register

Bit	Contents
7	
6	
5	— N/A
4	
3	
2	— Mask bit (Clear=0, Set=1)
1	
0	— Select Channel

SELECT CHANNEL

These two bytes indicate the channel select as follows:

- 0 — Channel 0
- 1 — Channel 1
- 2 — Channel 2
- 3 — Channel 3

Write All Mask Register

Bit	Contents
7	
6	
5	— N/A
4	
3	— Channel 3 (Enable=0)
2	— Channel 2 (Enable=0)
1	— Channel 1 (Enable=0)
0	— Channel 0 (Enable=0)

Mode Register

Bit	Contents
7	— Mode
6	
5	— Address
4	— Auto-initialize (Enable=1)
3	— Function
2	
1	— Select Channel
0	

MODE

These two bits indicate the mode as follows:

- 0 — Demand mode
- 1 — Single mode
- 2 — Block mode
- 3 — Cascade mode

ADDRESS

This bit is used to change the address as follows:

- 0 — Increment
- 1 — Decrement

FUNCTION

These two bits indicate the function as follows:

- 0 — Verify function
- 1 — Write function
- 2 — Read function

SELECT CHANNEL

These two bytes indicate the channel select as follows:

0 — Channel 0

1 — Channel 1

2 — Channel 2

3 — Channel 3

SEQUENCING AND TIMING

When the system is powered-up, it is recommended that all mode registers to be set with valid data even if the channel is not used.

Before loading the address and count registers, disable the controller (BIT 2 of COMMAND REGISTER) or mask the channel. This prevents erroneous transfers before a complete address is loaded.

A write to the Clear Flip/Flop sets the controller so that an access to an address or count is to the upper and lower byte in the correct sequence.

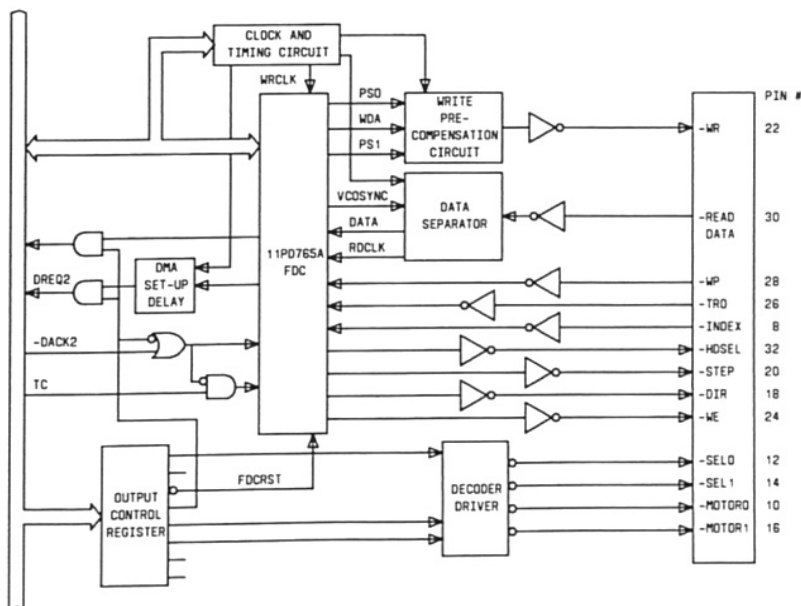
Floppy Diskette Controller (FDC)

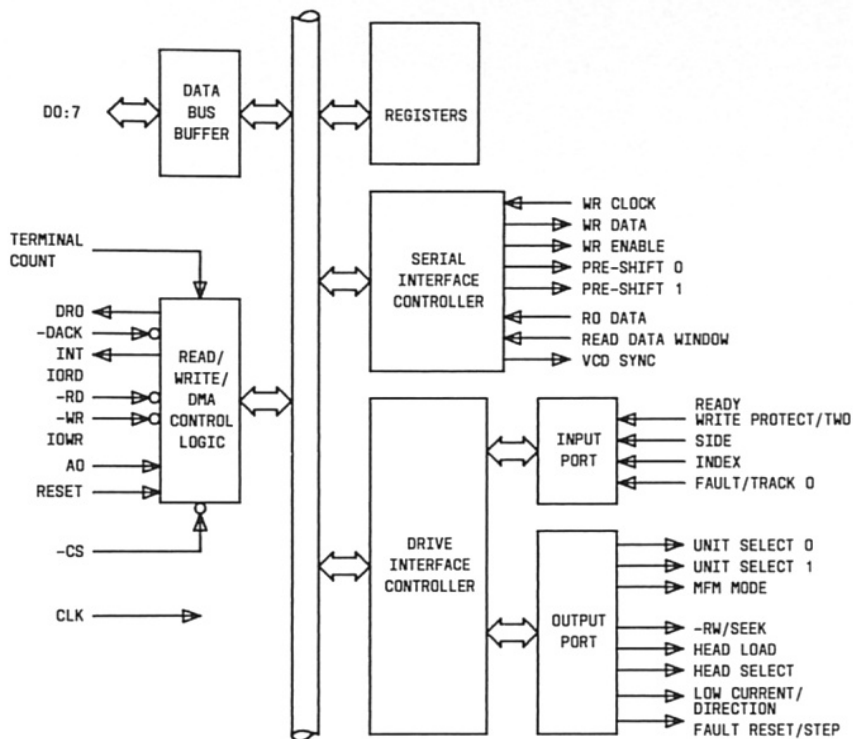
The diskette interface and NEC uPD765 controller read and write 5-¼ inch diskettes on as many as two drives. High-density or double-density formats are supported. High-density diskettes contain 1,228,800 bytes and double-density diskettes contain 368,640 bytes.

Each sector on the diskette contains an ID field and the Data field. The ID field contains the cylinder number, the head number, the sector number, and the number of bytes per sector.

The diskette controller performs 15 separate functions. It operates in either DMA or non-DMA mode. Interrupts can be enabled on INT6.

Block Diagram





REGISTERS

PORT	BYTE#	NAME
3F2	1	Interface Output Control
3F4	1	FDC Main Status
3F5	2	Status Register 0
3F5	3	Status Register 1
3F5	4	Status Register 2
3F5	5	Status Register 3
3F5	1	FDC Data
065H	1	FDC Speed Select

FDC PARAMETERS

Symbol	Name	Reference
DTL	Data Length	See following
EOT	End-of-Track	See following
GPL	Gap Length	See following
HLT	Head Load Time	Specify command
HUT	Head Unload Time	Specify command
N	Bytes/Sector	See following
MD	Mode	See following
FT	Format	See following
SRT	Step Rate Time	Specify command
STP	Scan Test Flag	See following
USx	Unit Select	See following

Data Length Number of bytes to read or write. Only applies when there are 128 bytes/sector (**N** = 0). Otherwise, **DTL** = FFH.

End-of-Track Last sector number on cylinder. If there are eight sectors/cylinder, then **EOT** = 8.

Gap Length The Gap Length between sectors. The **GPL** is different for read, write, and format commands. This table suggests appropriate values (hexadecimal).

Format (Density)	Sector Size	N	C	GPL(1)	GPL(2)
High	256	01	1A	0E	36
High	512	02	0F	1B	54
High	1024	03	08	35	74
High	2048	04	04	99	FF
High	4096	05	02	C8	FF
High	8192	06	01	C8	FF
Double	256	01	12	0A	0C
Double	256	01	10	20	32
Double	512	02	08	2A	50
Double	1024	03	04	80	F0
Double	2048	04	02	C8	FF
Double	4096	05	01	C8	FF

GPL(1) - Suggested GPL in read and write commands

GPL(2) - Suggested GPL in format commands

FORMAT

The type of format as follows:

- 0 — Single density (not implemented)
- 1 — Double (high) density

BYTES/SECTOR

The number of bytes per sector as follows:

- 0 — 128 bytes (not supported)
- 1 — 256 bytes
- 2 — 512 bytes
- 3 — 1024 bytes

MODE

The mode as follows:

- 0 — DMA Mode
- 1 — Non-DMA Mode

SCAN TEST FLAG

The type of comparisons for a scan test as follows:

- 1 — Sector-by-sector
- 2 — Alternate sectors

UNIT SELECT

The unit selection where "x" denotes the drive (0 or 1) as follows:

- US0 = 0 — Drive 0 not selected
- US0 = 1 — Drive 0 selected
- US1 = 0 — Drive 1 not selected
- US1 = 1 — Drive 1 selected

REGISTER FORMATS

Interface Output Control Register (Write Only)

Bit	Contents
7	—
6	— N/A
5	— Drive 1 motor (Off=0, On=1)
4	— Drive 0 motor (Off=0, On=1)
3	— FDC interrupt and DMA (Enable=1)
2	— FDC (Reset=0)
1	— N/A
0	— Select Drive (Drive 0=0, Drive 1=1)

FDC Speed Select Register (Write Only)

DATA	TYPE OF DRIVE	DATA TRANSFER RATE (kilobits/second)
00H	1.2M byte, high density	500
01H	1.2M byte, double density	300
02H	360K byte	250

FDC Main Status Register (Read Only)

Bit	Contents
7	— Data register ready to send or receive data (No=0, Yes=1)
6	— Transfer Direction
5	— FDC Phase on non-DMA operation
4	— FDC Busy (No=0, Yes=1)
3	—
2	— N/A
1	— Drive 1 Busy Seeking (No=0, Yes=1)
0	— N/A

TRANSFER DIRECTION

This bit indicates the direction of the data transfer as follows:

- 0 — From the FDC to the data register
- 1 — From the data register to the FDC

FDC PHASE ON NON-DMA OPERATION

This bit indicates the phase of the FDC as follows:

- 0 — Result phase
- 1 — Execution phase

FDC BUSY

The FDC is busy when a read or write command is in process.

Status Register 0 (Read Only)

Bit	Contents
7	— Status
6	—
5	— Seek complete (Yes=1)
4	— Equipment Check
3	— FDC Ready (Yes=0)
2	— Head selected at interrupt
1	— Drive 1 selected at interrupt (No=0, Yes=1)
0	— Drive 0 selected at interrupt (No=0, Yes=1)

STATUS

These two bits indicate the status as follows:

- 0 — Normal completion
- 1 — Abnormal completion
- 2 — Invalid command
- 3 — Drive not ready after start of command

EQUIPMENT CHECK

This bit indicates a fault condition on the drive or track 0 not found during recalibration as follows:

- 0 — No fault condition present
- 1 — Fault condition present

FDC READY

This bit indicates if the FDC is ready as follows:

- 0 — Ready
- 1 — Not ready

HEAD SELECTED AT INTERRUPT

This bit indicates the head selected at interrupt as follows:

- 0 — Head 0
- 1 — Head 1

Status Register 1 (Read Only)

Bit	Contents
7	— End of Cylinder (=1)
6	— N/A
5	— Data error in field (Yes=1)
4	— Data Overrun
3	— N/A
2	— No data
1	— Write Protect Signal
0	— ID Address Mark not Detected (Yes=1)

END OF CYLINDER

This bit indicates that the sector requested is beyond the end of the cylinder.

DATA ERROR IN FIELD

This bit indicates an error has occurred in the data or ID Field.

DATA OVERRUN

This bit indicates that the FDC was not serviced during data transfer within certain time when bit = 1.

NO DATA

This bit indicates one of the following conditions when bit = 1. See "Scan Commands" heading for more information.

- The data sector was not found during a read data, write deleted data, or scan command when bit = 1.
- There was a failure to read the sector ID during read ID command.
- The starting sector was not found during the read cylinder command.

WRITE PROTECT SIGNAL

This bit indicates a write protect signal has occurred when bit = 1.

Status Register 2 (Read Only)

Bit	Contents
7	— N/A
6	— Control mark (=1)
5	— Data error in field (Yes=1)
4	— Wrong cylinder (Yes=1)
3	— Equal condition satisfied during scan command (Yes=1)
2	— Scan not satisfied (Yes=1)
1	— Bad cylinder (Yes=1)
0	— FDC Cannot Find Mark

CONTROL MARK

This bit indicates that a deleted data address mark was encountered during a read or scan when bit = 1.

WRONG CYLINDER

This bit indicates that the cylinder or diskette is not what it should be when bit = 1.

BAD CYLINDER

This bit indicates that the cylinder or diskette is incorrect when data = 1.

FDC CANNOT FIND MARK

This bit indicates that the FDC cannot find a data address mark or a deleted data address mark during a read when the bit = 1.

Status Register 3 (Read Only)

Bit	Contents
7	— Fault on Drive (Yes=1)
6	— Drive Write Protected (Yes=1)
5	— Drive Ready (Yes=1)
4	— Track 0? (Yes=1)
3	— Two Side
2	— Selected Head
1	— Drive 1 Selected (Yes=1)
0	— Drive 0 Selected (Yes=1)

TWO SIDE

This bit indicates one or two sided diskette drive. This is not implemented in this application.

SELECTED HEAD

This bit indicates which head is selected as follows:

- 0 — Head 0
- 1 — Head 1

Data Register (Read & Write)

Command Code Byte

Bit	Contents
7	— Multi-track (MT)
6	— Format (Single=0, Double=1)
5	— Skip (SK)
4	
3	
2	— Command
1	
0	

This is the format of the first byte written to Data register to begin an instruction. One to eight more bytes must be written depending on the command used.

MT — MULTI-TRACK

This bit indicates whether or not an FDC is multi-track. When the bit is 1, the FDC will continue on side 1 after completing an operation on side 0.

SK — SKIP

This bit indicates whether or not to skip the sector with the Deleted Data Address mark and read the next one. When the bit is 1, the FDC will skip the sector.

COMMAND

These bits indicate the command.

The following heading, **Data Register Commands**, describes each command and provides the formats for the additional bytes.

Data Register Commands

The following commands apply to the Data register:

Code	Command
2	— Read Track (2 bytes)
3	— Specify
4	— Sense Drive Status
5	— Write Data
6	— Read Data
7	— Recalibrate
8	— Sense Interrupt Status
9	— Write Deleted Data
10	— Read ID
12	— Read Deleted Data
13	— Format a track
15	— Seek
17	— Scan Equal
25	— Scan Low or Equal
29	— Scan High or Equal

READ TRACK

This command reads all data fields from the index hole to EOT. The FDC continues reading even if it finds a CRC error in the ID or data fields.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

SPECIFY

This command defines the internal timers of the drive parameters.

The second byte has this format:

Bit	Contents
7	— Step Rate Time (SRT)
6	
5	
4	
3	— Head Unload Time (HUT)
2	
1	
0	

Step Rate Time (SRT) The amount of time to move the head from track to track. With a 4-MHz clock (360K byte drive), the SRT is set from 2 to 32 msec in 2 msec decrements as follows:

- 1 — 32 msec
- 2 — 30 msec
- 3 — 28 msec
- ·
- ·
- ·
- F — 2 msec

With an 8-MHz clock (1.2M byte drive), the SRT is set from 1 to 16 msec in 1 msec decrements as follows:

1 — 16 msec
2 — 15 msec
3 — 14 msec
.
.
.
F — 1 msec

Recommended SRT's are:

360K byte drive — 6 msec
1.2M byte drive — 4 msec

Head Unload Time (HUT)

The amount of time to wait after a read or write before the heads are unloaded. With a 4-MHz clock, the HUT is set from 0 to 480 msec in 32 msec increments as follows:

1 — 32 msec
2 — 64 msec
.
.
.
F — 480 msec

Note: If a new read or write command is issued quickly for the same track, before the head unloads, the head setting time will be eliminated.

The third byte has this format:

Bit	Contents
7	
6	
5	
4	— Head Load Time (HLT)
3	
2	
1	
0	— DMA Mode (Yes=0, No=1)

Head Load Time (HLT) The amount of time to load the heads before a read or write. With a 4-MHz clock, the HLT is set from 4 to 508 msec in 4 msec increments as follows:

1	— 4 msec
2	— 8 msec
3	— 12 msec
.	.
.	.
.	.
7F	— 508 msec

SENSE DRIVE STATUS

This command obtains the current drive status. The results of a Sense Drive Status command can be interpreted via Status Register 3.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

WRITE DATA

This command writes data.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

READ DATA

This command reads data from the diskette.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

RECALIBRATE

This command positions the head to track 0.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— N/A
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

Note: It is mandatory to follow this command with the Sense Interrupt Status command because an interrupt is generated.

SENSE INTERRUPT STATUS

This command is used to identify the cause of an unknown interrupt.

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 1
2	— 0
1	— 0
0	— 0

Status Register 0 indicates why the interrupt occurred.

WRITE DELETED DATA

This command writes deleted data.

The Write Deleted Data is the same as Write Data except that the FDC writes a Deleted Data Address mark at the beginning of the Data Field instead of the normal Data Address Mark. When reading deleted data, the FDC sets the CM error in Status Register 2 and reads the data. A Read Data would not read the data. If SK (byte 1) = 1, then the FDC skips the sector with the Deleted Data Address mark and reads the next one.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

READ ID

This command reads the first correct sector ID field.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

READ DELETED DATA

This command reads deleted data.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

FORMAT A TRACK

This command formats an entire track. The ID Field for each sector is supplied by the programmer during the execution phase.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

Note: Command 17 must be used before formatting to set the appropriate data transfer rate for the drive being formatted. This is **very** important since a high density diskette can be ruined by formatting it as a low density diskette.

SEEK

This command positions the head at the requested track.

When a seek is requested, the FDC checks its current position and decides in which direction to move. Then step pulses are issued to move the heads. The speed of the pulse is controlled by the Step Rate Time parameter in the **Specify** command. While drive 1 is seeking, bit 1 (Drive 1 Busy Seeking) in the Main Status Register is set (=1). It must be cleared by a Sense Interrupt Status command after the completion interrupt. While a drive is seeking, the FDC is not busy. Another Seek command to the other drive can be requested.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

SCAN EQUAL

This command scans for an equal data compare.

The second byte has this format:

Bit		Contents
7	—	0
6	—	0
5	—	0
4	—	0
3	—	0
2	—	Head Number (Head 0=0, Head 1=1)
1	—	Select Drive 1 (No=0, Yes=1)
0	—	Select Drive 0 (No=0, Yes=1)

SCAN LOW OR EQUAL

This command scans for a low or equal data compare.

The second byte has this format:

Bit		Contents
7	—	0
6	—	0
5	—	0
4	—	0
3	—	0
2	—	Head Number (Head 0=0, Head 1=1)
1	—	Select Drive 1 (No=0, Yes=1)
0	—	Select Drive 0 (No=0, Yes=1)

SCAN HIGH OR EQUAL

This command scans for a high or equal data compare.

The second byte has this format:

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— Head Number (Head 0=0, Head 1=1)
1	— Select Drive 1 (No=0, Yes=1)
0	— Select Drive 0 (No=0, Yes=1)

READ AND WRITE COMMANDS

If a read or write command terminates in error, then the values for cylinder, head, sector, and number of bytes per cylinder depend on the state of MT and EOT.

SCAN COMMANDS

The scan commands terminate when a scan condition is met, last sector on the track is reached, or a terminal count is received. The DMA issues the terminal count when it has no more data to send or receive. Determine the results of the scan via bits 2 and 3 (No data parameter) of the Status Register 2 as follows:

Command	No Data	Comments
Scan Equal	2	Diskette Data=Processor Data
Scan Equal	1	Diskette Data>Processor Data
Scan Low or Equal	2	Diskette Data=Processor Data
Scan Low or Equal	0	Diskette Data<Processor Data
Scan Low or Equal	1	Diskette Data>Processor Data
Scan High or Equal	2	Diskette Data=Processor Data
Scan High or Equal	0	Diskette Data>Processor Data
Scan High or Equal	1	Diskette Data<Processor Data

Scans allow the compare to be on contiguous sectors (STP=1) or alternate sectors (STP=2). However, for normal termination of the command, the last sector on the track must be compared.

SEQUENCING AND TIMING

There are three phases to each function:

- Command — The programmer writes the required information to the FDC.
- Execution — The FDC performs the operation.
- Result — The programmer reads the FDC's status.

COMMENTS

Before any data can be read or written to the FDC, the Main Status Register (MSR) must be read to determine the status of Bit 6 (Transfer Direction) and Bit 7 (Data Register Ready). In the command phase, Bit 6 must be 0 and Bit 7 must be 1. In the result phase, both bits must be 1. You must wait 12 usec after a data read or write before reading the MSR.

COMMAND PHASE

In the command phase, all output must be written (1 to 9 bytes). In the result phase, all status information must be read (1 to 7 bytes).

EXECUTION PHASE

During the execution phase of read and write commands, the following occurs:

- The heads are loaded if unloaded.
- The FDC waits for the head settle time to elapse.
- The FDC begins reading the ID address marks and ID field.
- When the requested sector number compares with the one of the diskette, the transfer begins.
- After completion of the transfer, the FDC waits the head unload time before unloading the heads.
- The amount of data that can be transferred in one instruction depends on the **Multi-track**, **MODE** and **N** parameters.

MT	Mode	Bytes/Sector	Maximum Transfer
		N	(Bytes/Sector)(Number of Sectors)
0	0	00	$128 \times 26 = 3,328$
0	1	01	$256 \times 26 = 6,656$
1	0	00	$128 \times 52 = 6,656$
1	1	01	$256 \times 52 = 13,312$
0	0	01	$256 \times 15 = 3,840$
0	1	02	$512 \times 15 = 7,680$
1	0	01	$256 \times 30 = 7,680$
1	1	02	$512 \times 30 = 15,360$
0	0	02	$512 \times 8 = 4,096$
0	1	03	$1024 \times 8 = 8,192$
1	0	02	$512 \times 16 = 8,192$
1	1	03	$1024 \times 16 = 16,384$

During the execution phase, the FDC operates in DMA mode or non-DMA mode. In DMA mode, there is one interrupt at the end of the phase. In non-DMA mode, there is an interrupt after the transfer of each byte.

In the Format a Track command, the ID field information for all the sectors in a track is sent to the FDC (cylinder, head, sector and bytes/sector). In DMA mode, 4 DMA requests per sector are issued. In non-DMA mode, there are 4 interrupts per sector.

If interrupts or DMA requests cannot be handled every 27 msec in double-density format or every 13 msec in high-density format during data transfers, the FDC sets the overrun error flag and terminates the operation.

When it is not executing a command, the FDC polls the drives looking for a change in drive ready. If there is a change, the FDC interrupts. You can determine the cause of

the unexpected interrupt with the Sense Interrupt Status Command.

Note: The drive motors should be off when the drives are not in use. However, they must be on prior to a drive select.

INTERRUPTS Interrupts occur as the result of:

- 1) Entering result phase of:
 - Read Data
 - Read Track
 - Read Deleted Data
 - Write Data
 - Write Deleted Data
 - Format Track
 - Scans
- 2) The execution phase in non-DMA mode
- 3) The Drive Ready line changing state
- 4) The end of Seek or Recalibrate

When the latter two occur, a Sense Interrupt Status determines the cause of the interrupt.

MT	HD	LAST SECTOR	ID INFORMATION RESULTS			
		TRANSFERRED	C	H	S	N
		EOT				
0	0	Less than EOT	NC	NC	S+1	NC
0	0	Equal to EOT	C+1	NC	S=1	NC
0	1	Less than EOT	NC	NC	S+1	NC
0	1	Equal to EOT	C+1	NC	S=1	NC
1	0	Less than EOT	NC	NC	S+1	NC
1	0	Equal to EOT	NC	LSB	S=1	NC
1	1	Less than EOT	NC	NC	S+1	NC
1	1	Equal to EOT	C+1	LSB	S=1	NC

NC = No Change

LSB = Least Significant BIT

Hard Disk Unit Controller

The Hard Disk Unit Controller reads and writes a maximum of two disk drives. The sector size is 512 bytes. The sectors can be interleaved in 16 different ways.

The Hard Disk Unit Controller operates in DMA or non-DMA mode. Interrupts can be enabled on INT5.

Extensive diagnostics are implemented. If a correctable data error is discovered, the error is automatically corrected using error correcting code (ECC).

REGISTERS

PORT	OPERATION	NAME
320	Read	Completion Status
320	Read/Write	Data
321	Write	Reset Controller
321	Read	Status
322	Write	Select Controller
322	Read	Drive Type
323	Write	Control

REGISTER FORMATS

Completion Status Register (Read Only)

Bit	Contents
7	— N/A
6	— N/A
5	— LUN (Drive 0=0, Drive 1=1)
4	— N/A
3	— N/A
2	— N/A
1	— Error (Yes=1)
0	— N/A

Control Register (Write Only)

Bit	Contents
7	
6	— N/A
5	
4	
3	— N/A
2	— N/A
1	— Enable interrupt when function completes (Yes=1)
0	— DMA transfer of data (Enable=1)

Status Register (Read Only)

Bit	Contents
7	
6	— N/A
5	— Request Interrupt (Yes=1)
4	— DREQ controller is requesting data to or from CPU (Yes=1)
3	— Controller (Selected=1)
2	— Command/Data byte selection
1	— Direction
0	— Controller Ready? (Yes=1)

COMMAND/DATA BYTE SELECTION

This bit indicates byte selection as follows:

- 0 — Data bytes
- 1 — Command or Status bytes

DIRECTION

This bit indicates the direction of I/O as follows:

- 0 — Data from CPU to controller
- 1 — Data from controller to CPU

Drive Type Register (Read Only)

Bit	Contents
7	— N/A
6	
5	
4	
3	— Drive 1 type
2	
1	— Drive 0 type
0	

This register is combined with the switches 3 and 4 at port 67H located on the motherboard (DIPSWITCH number 1 of motherboard). Switch 4 is associated with drive 0 and switch 3 with drive 1. The drive type is as follows:

Motherboard Setting	Drive Setting	Drive Type
ON	00	10M byte drive
ON	01	30M byte (CDC Wren) drive
ON	10	20M byte (CMI 6426) drive
ON	11	40M byte (Tandon) drive
OFF	00	40M byte (Segate ST405) drive
OFF	01	80M byte (Miniscribe 6086) drive
OFF	10	67M byte (Micropolis 1325) drive
OFF	11	20M byte (Miniscribe/Segate) drive

FUNCTIONS

The Hard Disk Unit Controller can perform these function codes:

Code	Transfer and Status Function
00	Test Drive Ready
01	Recalibrate
03	Read Status of Last Operation
04	Format Drive
05	Read Verify
06	Format Track
07	Format Bad Track
08	Read
0AH	Write
0BH	Seek
0CH	Initialize Drive Characteristics
0DH	Read ECC Burst Error Length
0EH	Read Sector Buffer
0FH	Write Sector Buffer

Code	Diagnostic Functions
E0H	Execute Sector Buffer Diagnostics
E3H	Execute Drive Diagnostic
E4H	Execute Controller Diagnostic
E5H	Read Long
E6H	Write Long

Format

All of the Hard Disk Unit Controller functions have this general format:

Byte	7	6	5	4	3	2	1	0
1	Function Code							
2	LUN			Pri Head#				
3	Cyl Hi		Sector#					
4	Cyl Low							
5	# of Blocks / Interleave							
6	Control							

Parameters The format consists of these parameters:

Parameter	Description
Function Code	Contains the code of the function
Pri Head#	Specifies the disk head number — valid range is 0 through 8
Cyl Hi/Low	Specifies the cylinder number which is 10 bits wide — contains the upper two bits
Sector#	Specifies the sector number — valid range is dependent on the track format
# of Blocks	Specifies number of sectors to transfer.
Interleave	Specifies the interleave for the format. A block count of 0 equals 256.

Note: The maximum interleave equals
Number of sectors/track - 1. The
maximum is 16 in this application.

Control — Byte Format

This byte tells the controller how to react to errors and defines the step mode. The step mode bits only apply to certain command execution phases. The control byte format is:

Bit	Contents
7	— Retry (Disable=1)
6	— Reread (Disable=1)
5	— 0
4	— 0
3	— 0
2	
1	— Step Mode
0	

Retry

This controls the retry for all errors except a Data ECC error. When this bit is "0", no retries are attempted. When this bit is "1", up to 10 retries are attempted.

Reread

When this bit is "1", an attempt is made to correct the error on the first syndrome. **When this bit is "0"**, there must be two consecutive like syndromes before an attempt is made to correct the error.

Step Mode

These bits indicate the step mode as follows:

- 0 — 3 msec
- 1 — 3 msec
- 2 — 3 msec
- 3 — 3 msec
- 4 — 200 usec
- 5 — 70 usec
- 6 — 3 msec
- 7 — 3 msec

**Test Drive
Ready
Function**

This function selects the specified drive and verifies the drive is ready, the seek is complete, and there are no drive faults. Electrically, this command tests the **DRDY**, **WF**, and **SC** signals of the selected drive. If **WF** and **SC** are low and **DRDY** is high, the command returns an error code of "0" (no error).

The Test Drive Ready function has this format:

Byte	7	6	5	4	3	2	1	0
1	0							
2	LUN			0				
3	0							
4	0							
5	0							
6	0							

Recalibrate Function

This function positions the read/write arm at track 0 and clears errors in the drive.

The Recalibrate function has this format:

Byte	7	6	5	4	3	2	1	0
1	1							
2	LUN			0				
3	0							
4	0							
5	0							
6	Control							

Request Sense Function

This function sends the four Sense Bytes to the CPU as data.

The Request Sense function has this format:

Byte	7	6	5	4	3	2	1	0
1	3							
2	LUN				0			
3	0							
4	0							
5	0							
6	0							

After this function is performed, the controller will collect the following:

Byte	7	6	5	4	3	2	1	0
1	AV	0	Error Code					
2	LUN			Head#				
3	Cyl Hi		Sector#					
4	Cyl Low							

AV — Address Valid

This bit indicates that the head, cylinder, and sector fields are valid.

Error Code

The error codes (all values in hexadecimal) are as follows:

Code	Type of Error
00	No error
02	No seek complete from drive
03	Write fault
04	Drive not ready
06	Track 0 not found
08	Drive still seeking
11	Uncorrectable data error
12	Address mark not found
15	Seek error
18	Correctable data error
19	Track is marked bad
20	Invalid command
21	Illegal sector address
30	Sector buffer error
31	Controller ROM checksum error
32	ECC polynomial error

**Format
Drive
Function**

This function formats all the tracks starting with the one specified in the function block to the end of the drive. The selected track format is used. The sectors are placed on the tracks according to the interleave code. The data fields are filled with the data pattern from the sector buffer.

The Format Drive function has this format:

Byte	7	6	5	4	3	2	1	0				
1	4											
2	LUN				Head#							
3	Cyl Hi		V									
4	Cyl Low											
5	Interleave											
6	Control											

Note: The parameter **V**, in byte 3, is not used but must be within a valid range.

**Read Verify
Function**

This function reads the specified number of blocks but does not transfer the data to the CPU. The function specifies the sector number where verification begins.

The Read Verify function has this format:

Byte	7	6	5	4	3	2	1	0				
1	5											
2	LUN				Head#							
3	Cyl Hi				Sector#							
4	Cyl Low											
5	# of Blocks											
6	Control											

Format Track Function

This function formats the specified track with no flags set in the ID fields. It fills the data field with the data pattern in the sector buffer. The interleave should be the same for the entire drive.

The Format Track function has this format:

Byte	7	6	5	4	3	2	1	0
1	6							
2	LUN				Head#			
3	Cyl Hi		0					
4	Cyl Low							
5	Interleave							
6	Control							

Format Bad Track Function

This function formats the track with the bad block flag set in all ID fields. It fills the data field with the data pattern in the sector buffer. The interleave must be the same for the entire drive.

The Format Bad Track function has this format:

Byte	7	6	5	4	3	2	1	0
1	7							
2	LUN				Head#			
3	Cyl Hi		0					
4	Cyl Low							
5	Interleave							
6	Control							

**Read
Function**

This function reads the specified number of blocks. The function specifies the initial sector address. The data is transferred to the CPU.

The Read function has this format:

Byte	7	6	5	4	3	2	1	0
1	8							
2	LUN				Head#			
3	Cyl Hi		Sector#					
4	Cyl Low							
5	# of Blocks							
6	Control							

**Write
Function**

This function writes the data starting at the initial block address given in the function.

The Write function has this format:

Byte	7	6	5	4	3	2	1	0
1	A							
2	LUN			Head#				
3	Cyl Hi			Sector#				
4	Cyl Low							
5	# of Blocks							
6	Control							

Seek Function

This function seeks to the cylinder of the specified block. For Winchester drives capable of overlap seeks, this function returns completion status before the seek is complete.

The Seek function has this format:

Byte	7	6	5	4	3	2	1	0				
1	B											
2	LUN				Head#							
3	Cyl Hi				V							
4	Cyl Low											
5	0											
6	Control											

Initialize Drive Characteristics Function

This function sets up the drive with different capacities and characteristics.

The Initialize Drive Characteristics function has this format:

Byte	7	6	5	4	3	2	1	0
1	C							
2	LUN			0				
3	0							
4	0							
5	0							
6	0							

After the function is performed, the controller will collect the following:

Byte	7	6	5	4	3	2	1	0
1	Max # of Cyl Hi							
2	Max # of Cyl Low							
3	Max # of Heads							
4	Reduced WR. CUR. Cyl Hi							
5	Reduced WR. CUR. Cyl Low							
6	Write Precomp. Cyl Hi							
7	Write Precomp. Cyl Low							
8	Max ECC Data Burst Length							

Read ECC Burst Error Length Function

This function transfers one byte of data to the CPU. This byte contains the ECC burst length that the controller detected for the correctable ECC data error during the last read function.

The Read ECC Burst Error Length function has this format:

Byte	7	6	5	4	3	2	1	0
1	D							
2	0							
3	0							
4	0							
5	0							
6	0							

After this function is performed, the controller will collect the following:

Byte	7	6	5	4	3	2	1	0
1	ECC Burst Length							

Read Sector Buffer Function

This function reads the contents of the Hard Disk Unit Controller buffer. No data transfer occurs between the controller and the drives.

The Read Sector Buffer function has this format:

Byte	7	6	5	4	3	2	1	0
1					E			
2					0			
3					0			
4					0			
5					0			
6					0			

Write Sector Buffer Function

This function writes one sector's worth of data to the controller sector buffer. No data transfer occurs between the controller and the drives.

The Write Sector Buffer function has this format:

Byte	7	6	5	4	3	2	1	0
1					F			
2					0			
3					0			
4					0			
5					0			

**RAM
Diagnostic
Function**

This function performs a data pattern test on the controller RAM.

The RAM Diagnostic function has this format:

Byte	7	6	5	4	3	2	1	0
1	E0							
2	0							
3	0							
4	0							
5	0							
6	0							

**Drive
Diagnostic
Function**

This function performs a diagnostic on the specified unit. It reads sector 1 on sequential tracks and then reads sector 1 on each track verifying ID and DATA fields.

The Drive Diagnostic function has this format:

Byte	7	6	5	4	3	2	1	0
1	E3							
2	LUN			0				
3	0							
4	0							
5	0							
6	Control							

Controller Internal Diagnostic Function

This function performs the controller internal diagnostics. The controller checks the internal processor, data buffer, ECC circuit and the checksum.

The Controller Internal Diagnostic function has this format:

Byte	7	6	5	4	3	2	1	0
1	E4							
2	0							
3	0							
4	0							
5	0							
6	0							

Read Long Function

This function reads sectors of data and ECC bytes from the disk and transfers them to the CPU. If an ECC error occurs during the read, the controller does not attempt to correct the data.

The Read Long function has this format:

Byte	7	6	5	4	3	2	1	0
1	E5							
2	LUN				Head#			
3	Cyl Hi		Sector#					
4	Cyl Low							
5	# of Blocks							
6	Control							

That is, the controller transfers 512 bytes plus 4 ECC bytes to the CPU for each sector read.

**Write Long
Function**

This function writes blocks of data and ECC bytes from the CPU to the disk without generating ECC for the data.

The Write Long function has this format:

Byte	7	6	5	4	3	2	1	0
1	E6							
2	LUN				Head#			
3	Cyl Hi		Sector#					
4	Cyl Low							
5	# of Blocks							
6	Control							

That is, the controller transfers 512 bytes plus 4 ECC bytes to the CPU for each sector read.

**SEQUENCING
AND
TIMING**

There are three phases to each function:

- Function initiation
- Function execution
- Function results.

To initiate a function, you select the controller with the Select Controller Register. Then you wait for Ready in the Status Register to be set. The In/Out bit and the Command/Data bit should indicate function transfer to the controller. You then write six function bytes to the Data Register.

If the Ready is set after this transfer, either there was an error in the function bytes or the controller is ready to receive another group of six function bytes and/or data.

Data can be transferred in DMA or non-DMA mode. If the transfer is in DMA mode, the DMA Controller is programmed in Demand Transfer mode (see DMA Controller). The count word is set to:

(number of sectors to transfer)(bytes/sector)- 1

If data is transferred in non-DMA, you use Ready, In/Out, Command/Data and Interrupt Request to time the transfer during execution.

Execution begins when the last function byte is received. In data transfer functions, the controller temporarily stores the data in the sector buffer. This prevents data overruns. When the function completes and the Completion Status byte is loaded, the controller issues interrupts if requested.

Clear the Interrupt Enable and the DMA enable bits in the Control Register after reading the Completion Status. This allows the controller to clear Interrupt Request and Data Request in the Status Register. It also clears the Selected bit.

The controller does extensive error recovery. If an error is found, three retries are attempted. If a retry is successful, the error is not reported; however, the retry count is incremented.

The following errors result in a retry:

- Seek error
- Sector not found
- Uncorrectable data error
- Correctable data error
- No data address mark
- No ID address mark
- ECC error in ID field.

Note: On a seek error, recalibrate and reseek is done by the controller.

The following errors are accumulated in the log:

- ECC error in ID field
- Correctable error in data field
- Uncorrectable error in data field
- No ID address mark
- No data address mark
- Seek error
- Record not found.

If rereads are disabled, the controller does not reread before applying the ECC correction.

The interleave factor states how many physical sectors logical sectors are apart.

For example, if the Interleave Factor is 2 and there are 17 sectors in a track, then a sector looks like this:

Physical Sector	Logical Sector
0	0
1	9
2	1
3	10
4	2
5	11
6	3
7	12
8	4
9	13
10	5
11	14
12	6
13	15
14	7
15	16
16	8

The track layout for 512 bytes per sector, 17 sectors per track is:

14 bytes '00'	A 1	I D E N T	C Y L O W	H E A D	S E C #	C C 1	C C 2	3 bytes '00'	12 bytes '00'	A 1	F 8	USER DATA	4 ECC	3 bytes	GAP:3 4E 1
---------------------	--------	-----------------------	-----------------------	------------------	------------------	-------------	-------------	--------------------	---------------------	--------	--------	--------------	----------	------------	------------------

Example

This program prepares the controller to receive a function:

```
select      equ 322
status      equ 321
control     equ 323
init-ctrl:  mov dx,select    ;select port address
            out dx,al        ;output anything
            mov dx,control   ;control port address
            mov al,3h        ;enable interrupts and dma
            out dx,al
            mov dx,status    ;status port address
init1:      in al,dx         ;get status
            test al,1h       ;is it ready?
            loopz init1      ;loop if not ready
            cmp al,dh        ;jump if ready for a
            je init2         ;function and selected
            stc              ;flag error
            stc
            ret
init2:      clc
            ret
```

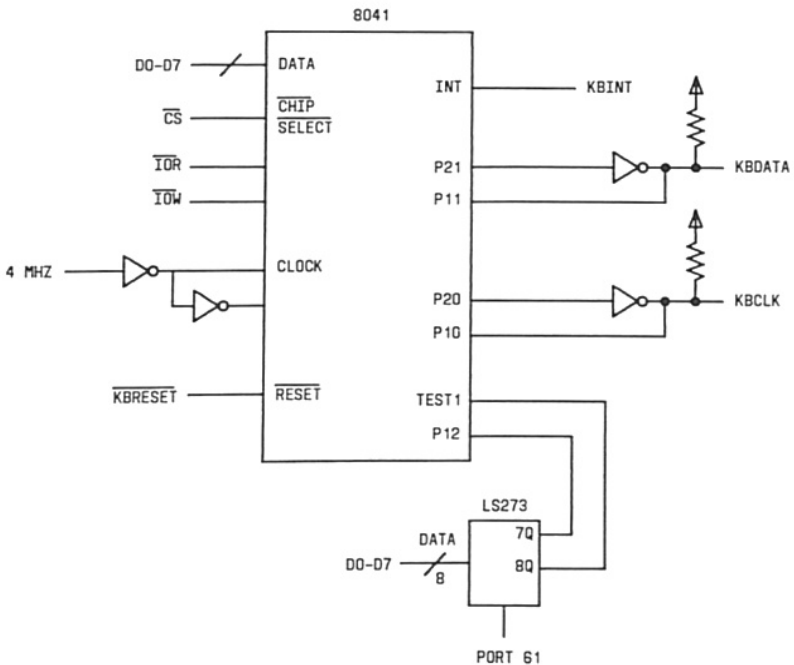
Keyboard Interface

The keyboard interface converts the parallel data into serial data for transmission to and from the keyboard. An INTEL 8041 microcontroller is used to control the keyboard interface.

To provide maximum flexibility in defining keyboard operations, the keyboard uses scan codes rather than ASCII codes. In addition, all keys generate a make scan code when pressed and a break scan code when released. The break scan code is 80H plus the make scan code.

The keyboard is responsible for keeping track of the amount of time a key is depressed and for generating the repeat key signal. All the keys have this repeat function.

Block Diagram



FUNCTIONS

The Keyboard Interface can perform the following functions:

- Allows you to determine when there is a character to read and when the keyboard is ready to receive a character via the Status register.
- Allows you to read the scan code of the key that was pressed or released via the Data register.
- Allows you to issue control commands to the keyboard via the Control register.
- Turns on either the CAPS LOCK, NUM LOCK, or SCROLL LOCK LED via the Data register.

REGISTERS

PORT	NAME	NOTES
60	Data	8-bit scan code when pressed, 8-bit scan code plus 80H when released. See scan code chart in Chapter 8.
61	Keyboard Control	
64	Status	

REGISTER FORMATS

Keyboard Control Register (Read & Write)

Bit	Contents
7	— Reset interrupt pending? (No=0, Yes=1)
6	— Keyboard Clock (Reset=0)
5	— I/O Channel (Enable=1)
4	— Parity Interrupt (Enable=1)
3	— N/A
2	— N/A
1	— Speaker pulse train (Yes=0)
0	— Speaker

SPEAKER PULSE TRAIN

See Speaker Interface

SPEAKER

This bit indicates whether the speaker is on or off as follows:

0 — Off

1 — On

Status Register (Read Only)

Bit	Contents
7	
6	
5	
4	— N/A
3	
2	
1	— 0 (OK to Write Byte)
0	— 1 (Byte to read)

SEQUENCING AND TIMING To send a character to the keyboard, wait for Bit 1 of the Status register to be set and write the byte to the data register.

To receive a character, wait for Bit 0 of the Status register to be set and read the character. The keyboard interface can be programmed to interrupt on INT1 when there is a character to read.

Example This program sets the CAPS LOCK LED on:

```
status      equ 64h
data        equ 60h
set_led:    in al,status      ;read status
            test al,2         ;keyboard ready to receive
            jnz set_led       ;input? Jump if ready.
            mov al,13h        ;keyboard LED command
            out data,al
set1:       in al,status      ;read status
            test al,2         ;ready to receive input?
            jnz set1          ;jump if not
            mov al,1          ;CAPS LOCK
            out data,al       ;write out code
            ret
```

KEYBOARD COMMANDS/RESPONSES Commands are sent one at a time from the system module to the Keyboard. The Keyboard receives data through the Keyboard command/status port location (64H).

Modify Status LED Command

Can change the state of the Caps Lock, Num Lock, and/or Scroll Lock LED indicators by entering the Modify Status LED command (13H) followed by a 1-byte parameter:

Byte	Description														
0	13H—Modify Status LED Command														
1	Command: <table> <tr> <th>Value</th><th>Function</th></tr> <tr> <td>01H</td><td>Caps Lock LED off</td></tr> <tr> <td>81H</td><td>Caps Lock LED on</td></tr> <tr> <td>02H</td><td>Num Lock LED off</td></tr> <tr> <td>82H</td><td>Num Lock LED on</td></tr> <tr> <td>04H</td><td>Scroll Lock LED off</td></tr> <tr> <td>84H</td><td>Scroll Lock LED on</td></tr> </table>	Value	Function	01H	Caps Lock LED off	81H	Caps Lock LED on	02H	Num Lock LED off	82H	Num Lock LED on	04H	Scroll Lock LED off	84H	Scroll Lock LED on
Value	Function														
01H	Caps Lock LED off														
81H	Caps Lock LED on														
02H	Num Lock LED off														
82H	Num Lock LED on														
04H	Scroll Lock LED off														
84H	Scroll Lock LED on														

Set Repeat Rate/Delay Command

This command (302 keyboard only) sets the delay interval and repeat rate used in the auto repeat function. The command has the following five-byte format.

Byte	Description																		
0	12H—Set Repeat Rate/Delay command																		
1	FEH—Constant to distinguish this command from the Set Mouse Mode 2 command																		
2	Delay value—Delay interval is determined by multiplying the delay value by 250 ms, e.g., 0CH yields a 3 s delay																		
3	Rate byte: <table> <tr> <th>Value</th><th>Rate</th></tr> <tr> <td>14H</td><td>4 Hertz</td></tr> <tr> <td>09H</td><td>8 Hertz</td></tr> <tr> <td>08H</td><td>10 Hertz</td></tr> <tr> <td>06H</td><td>12 Hertz</td></tr> <tr> <td>05H</td><td>16 Hertz</td></tr> <tr> <td>04H</td><td>20 Hertz</td></tr> <tr> <td>03H</td><td>24 Hertz</td></tr> <tr> <td>02H</td><td>40 Hertz</td></tr> </table>	Value	Rate	14H	4 Hertz	09H	8 Hertz	08H	10 Hertz	06H	12 Hertz	05H	16 Hertz	04H	20 Hertz	03H	24 Hertz	02H	40 Hertz
Value	Rate																		
14H	4 Hertz																		
09H	8 Hertz																		
08H	10 Hertz																		
06H	12 Hertz																		
05H	16 Hertz																		
04H	20 Hertz																		
03H	24 Hertz																		
02H	40 Hertz																		
4	FFH—Constant																		

Mouse (Optional)

The Mouse is designed to transfer and encode the distance that it is moved across a smooth surface. The user moves the Mouse over a table top, rotating the ball underneath. The ball movement is translated into X and Y movement by two perpendicular shafts activated by the ball. The motion of a shaft, sensed by optical decoders, causes the two output bits for that direction to form waves in quadrature, i.e., two square signals, which are out-of-phase by 90 degrees, are produced for each axis.

Frequency is determined by the speed and phase, $\pm 90^\circ$, is determined by the direction of travel. About 8 pulses per millimeter are provided, depending on the speed at which the Mouse is moved.

MOUSE OPERATION

The Mouse has two (or three) buttons that are used to trigger a particular action. They are: "left," "middle," and "right."

Scan Codes

The Mouse buttons default scan codes are listed in the following table.

Button	Scan Code
Left	1CH (carriage return key)
Middle (3-button Mouse only.)	53H (delete key)
Right	01H (escape key)


The scan codes for the Mouse buttons may be changed by either the Set Mouse Mode 1 command or the Set Mouse Mode 2 command from the System Module to the Keyboard. When a Mouse button is held down, the scan code is not automatically repeated. When the

button is released the Keyboard returns a scan code which is 80H greater than the programmed value.

Modes of Operation

The Keyboard interfaces the Mouse through two modes of operation. The modes differ in how the Keyboard reports the X (left and right) and Y (up and down) displacement of the Mouse to the System Module.

Mode 1

Mode 1 is the default mode and is set during either system power-up or after the Keyboard has been reset. In mode 1, the Keyboard determines the X and Y motion of the Mouse and divides it by a scale factor. Whenever the Mouse has been moved, a scan code, by default, corresponding to the "UP," "DOWN," "LEFT," or "RIGHT" arrow key, is sent. If the  LED is illuminated, then the direction codes are preceded by the shift depressed scan code and followed by the shift released scan code. The default scale factors are 2 for the X-axis and 4 for the Y-axis. The default scale factors and scan codes may be changed with the Set Mouse Mode 1 Command. To change the scale factors and button scan codes, you would output the following 10-byte code sequence to the keyboard.

Byte	Description
0	11H (Set Mouse
1	Left button scan code
2	Middle button scan code
3	Right button scan code
4	Left direction code
5	Right direction code
6	Down direction code
7	Up direction code
8	X-axis scaling factor
9	Y-axis scaling factor

The scan codes may be any value in the range of 0 to 7FH.

Mode 2

In mode 2, the X and Y motion reports are sent to the system every 20 ms as a signed 8-bit number, i.e., a range of -127 to 127. These X and Y values represent relative changes to previous X and Y positions. The format of the response from the Keyboard is as follows:

Byte	Description
0	FEH (Response indicator)
1	X Mouse displacement
2	Y Mouse displacement

The scale factor is not used in mode 2. When the counters overflow, the displacement is ignored and no overflow indication is provided. Negative displacement values are sent as ones complement numbers where minus one has a value of FEH.

To select Mode 2, you would send the following 5-byte code sequence to the Keyboard.

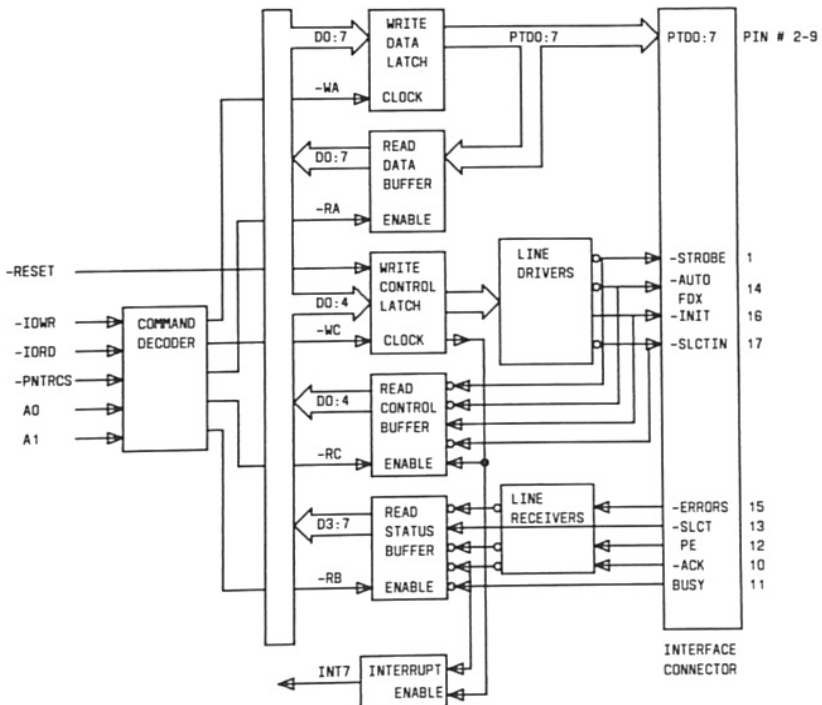
Byte	Description
0	12H (Set Mouse Mode 2 command)
1	Left button scan code
2	Middle button scan code
3	Right button scan code
4	Reserved (0H)

Parallel Printer Interface

The parallel printer interface connects to printers with a Centronics-like parallel interface or any other device with identical interface characteristics. The input and output signals are presented to the external device through a 25-pin "D" type connector.

The interface has five buffered outputs — data, strobe, initialize printer, automatic linefeed, and select. These can be read and written. In addition, the interface has five inputs — acknowledge, busy, paper out, error, and select. An interrupt can be enabled on INT7.

Block Diagram



FUNCTIONS

The Parallel Printer Interface can perform the following functions:

- Clear the data buffer and initialize the Control register after a power on or hardware reset. The Control register is initialized with bits 0 through 4 set to "0".
- Enable the data to be written to the printer data bus via the Data register. The actual writing occurs when the strobe line is activated.
- Invert D0, D1, and D3 on the data bus and writes the data to the control register. If D4 is "1" then interrupts are requested.
- Enable the data to be read to the data bus via the Data register. It normally is the last character written to the printer.
- Enable the data on the printer control lines and the interrupt control bit to be placed on the data bus via the Control register.
- Enable the data on the printer status lines to be placed on the data bus via the Status register.

REGISTERS

PORT	NAME	NOTES
378	Data	Print Character
379	Status	
37A	Control	

REGISTER FORMATS

Status Register (Read Only)

Bit	Contents
7	— Busy? (Yes=1)
6	— Character print (Yes=1)
5	— Out of paper? (Yes=1)
4	— Printer selected? (Yes=1)
3	— Error? (Yes=0)
2	
1	— N/A
0	

Control Register (Read & Write)

Bit	Contents
7	— Interrupts requested (Yes=0)
6	— Select? (Yes=0)
5	— Initialize (Yes=0)
4	— Auto linefeed (Yes=0)
3	— Strobe (Low=0, High=1)
2	
1	— N/A
0	

SEQUENCING AND TIMING

To send a character to the printer, the character is put on the data bus. When the printer is not busy, it is ready to accept the next character. The character must be strobed into the printer by setting the strobe bit 1 for at least five nanoseconds and the resetting it.

Interrupts can be enabled on INT7. The interrupt is requested when Bit 6 of the Status register goes to 0.

To initialize the printer, first select it. Then issue the initialize command setting the automatic line feed and interrupt parameters.

Example

This program sends characters to the printer and gets status information:

```
data          equ 378h
print_char:mov dx,data          ;get data port
              out dx,al         ;put char on data line
              inc dx            ;get status port
              in al,dx          ;read in status
              test al,080h      ;is the printer busy?
              jnz print_not_busy ;jump if not busy
              .
              .
              .
print_not_busy: mov al,0dh       ;strobe high
              inc dx            ;get control port
              out dx,al         ;to control register
              nop               ;wait
              nop               ;wait
              mov al,0ch        ;strobe low
              out dx,al         ;to control register
              dec dx            ;get status port
              in al,dx          ;read in status
              ret
```

Programmable Interrupt Controller (PIC)

The Intel 8259A Programmable Interrupt Controller (PIC) handles up to eight vectored priority interrupts for the CPU. It is cascadable for up to 64 vectored priority interrupts without additional circuitry.

The PIC can operate in the following modes:

- **Fully Nested Mode**

This is the default mode. The interrupt requests have an ordered priority from 0 (highest) to 7 (lowest). The highest priority is acknowledged first and those of lower priority are inhibited.

- **Special Mask Mode**

This mode is similar to fully-nested mode except that the interrupt mask register (IMR) determines which interrupts are disabled.

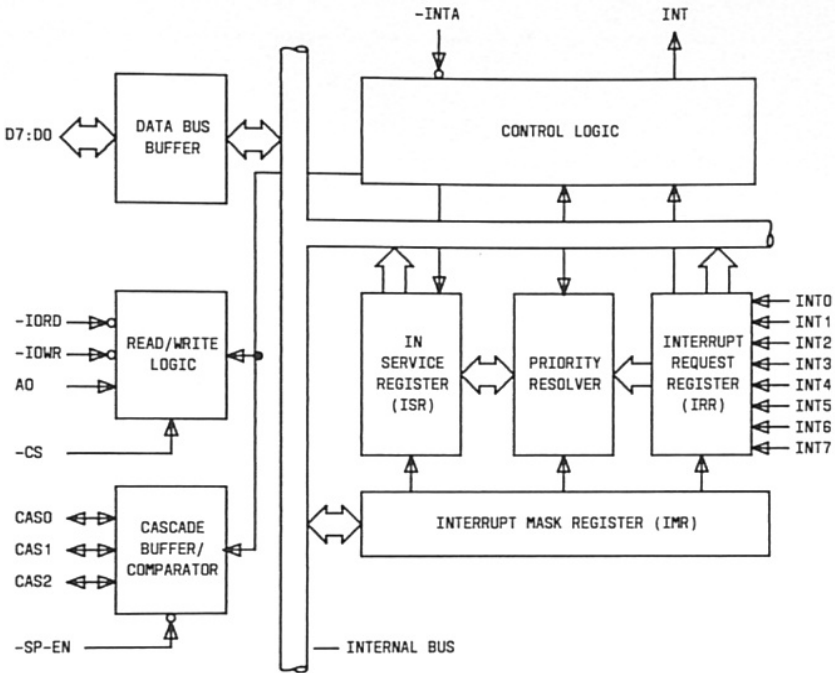
- **Poll Command Mode**

This mode allows the CPU to poll the devices. It is selected by disabling interrupts with the CLI instruction. Periodically the CPU polls the PIC to receive the interrupt type of the highest priority device requesting service.

- **Automatic Rotation**

In this mode, a device receives the lowest priority after it is serviced. All other devices have their priorities adjusted accordingly. The next highest interrupt line receives the highest priority.

Block Diagram



REGISTERS

PORT	BYTE#	NAME	
20	1	Initial Command Word 1	
21	1	Initial Command Word 2	INT0 interrupt type, multiple of eight
21	2	Initial Command Word 3	Cascade mode only
21	3	Initial Command Word 4	
21	4	Operation Command Word 1	Interrupt Mask register
20	2	Operation Command Word 2	
20	3	Operation Command Word 3	
20	4	In-Service	
21	5	Interrupt Level	
20	5	Interrupt Request	

REGISTER FORMATS

Initial Command Word 1 Register (Write Only)

Bit	Contents
7	
6	— N/A
5	
4	— 1
3	— Triggered Mode
2	— Call Address Interval
1	— Mode
0	— ICW4 needed (Yes=1)

TRIGGERED MODE

This bit indicates the type of triggered mode as follows:

- 0 — Edge triggered
- 1 — Level triggered

CALL ADDRESS INTERVAL

This bit indicates the size of the interrupt as follows:

- 0 — interval of 8
- 1 — interval of 4

MODE

This bit indicates the mode as follows:

- 0 — Cascade mode
- 1 — Single mode

ICW4 NEEDED

This bit indicates whether or not the Initial Command Word 4 register is needed as follows:

- 0 — No
- 1 — Yes

Initial Command Word 2 Register (Write Only)

Bit	Contents
7	— Interrupt Vector Address 7
6	— Interrupt Vector Address 6
5	— Interrupt Vector Address 5
4	— Interrupt Vector Address 4
3	— Interrupt Vector Address 3
2	
1	— N/A
0	

Initial Command Word 3 Register (Write Only) (Master Device)

Bit	Contents
7	— Device 7 slave (Yes=1)
6	— Device 6 slave (Yes=1)
5	— Device 5 slave (Yes=1)
4	— Device 4 slave (Yes=1)
3	— Device 3 slave (Yes=1)
2	— Device 2 slave (Yes=1)
1	— Device 1 slave (Yes=1)
0	— Device 0 slave (Yes=1)

Initial Command Word 3 Register (Write Only) (Slave Device)

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	
1	— Slave Device ID
0	

Initial Command Word 4 Register (Write Only)

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— Fully-nested Mode (Yes=1)
3	— Buffered
2	— Buffered
1	— 1=Auto EOI, 0=Normal EOI
0	— 1

BUFFERED

This bit indicates the buffered status as follows:

- 00 — Non-buffered (slave)
- 01 — Non-buffered (master)
- 10 — Buffered mode (slave)
- 11 — Buffered mode (master)

Operation Command Word 1 Register (Read & Write)

Bit	Contents
7	— Interrupt Mask 7 (Set=1)
6	— Interrupt Mask 6 (Set=1)
5	— Interrupt Mask 5 (Set=1)
4	— Interrupt Mask 4 (Set=1)
3	— Interrupt Mask 3 (Set=1)
2	— Interrupt Mask 2 (Set=1)
1	— Interrupt Mask 1 (Set=1)
0	— Interrupt Mask 0 (Set=1)

Operation Command Word 2 Register (Write Only)

Bit	Contents
7	— Specify EOI / Rotation
6	
5	
4	— 0
3	— 0
2	— Interrupt Level
1	
0	

SPECIFY EOI / ROTATION

These bits specifies the EOI as follows:

- 0 — Rotate in automatic EOI mode (clear)
- 1 — Non-specific EOI
- 2 — No operation
- 3 — Specific EOI
- 4 — Rotate in automatic EOI mode (set)
- 5 — Rotate on non-specific EOI
- 6 — Set priority command
- 7 — Rotate on specific EOI

INTERRUPT LEVEL

These bits indicate the interrupt level as follows:

- 0 — Level 0
- 1 — Level 1
- .
- .
- .
- 7 — Level 7

Operation Command Word 3 Register (Write Only)

Bit	Contents
7	— 0
6	— Special Mask Mode
5	—
4	— 0
3	— 1
2	— Poll command (Yes=1)
1	—
0	— Read Register command

SPECIAL MASK MODE

These bits indicate the special mask mode as follows:

- 0 — No action
- 1 — No action
- 2 — Reset special mask
- 3 — Set special mask

READ REGISTER COMMAND

These bits indicate the read request as follows:

- 0 — No action
- 1 — No action
- 2 — Read IRR on next read
- 3 — Read ISR on next read

In-Service Register (Read Only)

Bit	Contents
7	— INT7 in service? (Yes=1)
6	— INT6 in service? (Yes=1)
5	— INT5 in service? (Yes=1)
4	— INT4 in service? (Yes=1)
3	— INT3 in service? (Yes=1)
2	— INT2 in service? (Yes=1)
1	— INT1 in service? (Yes=1)
0	— INT0 in service? (Yes=1)

Interrupt Request Register (Read Only)

Bit	Contents
7	— INT7 request service? (Yes=1)
6	— INT6 request service? (Yes=1)
5	— INT5 request service? (Yes=1)
4	— INT4 request service? (Yes=1)
3	— INT3 request service? (Yes=1)
2	— INT2 request service? (Yes=1)
1	— INT1 request service? (Yes=1)
0	— INT0 request service? (Yes=1)

Interrupt Level Register (Read Only)

Bit	Contents
7	
6	
5	— N/A
4	
3	
2	
1	— Highest priority level requesting service (0 through 7)
0	

**SEQUENCING
AND
TIMING**

When the ICW1 command is issued, the initialization process begins. The following automatically occurs:

- IMR is cleared.
- INT7 is assigned the lowest priority.
- Single mode is assumed.
- Special Mask Mode is cleared.
- A status read fetches IRR.
- If Bit 0 equals zero, then ICW4 functions are set to zero.

Next ICW2 is output followed by ICW3. If the ICW4 was requested by ICW1, then it is output. This completes initialization.

Once the initialization process is complete, the PIC is ready to accept interrupts. Any of the functions to change the priority scheme can be executed. In addition, the IRR, IMR, and ISR can be read.

If automatic EOI is not specified, the interrupt service routine must declare EOI. Either specific or non-specific EOI can be used depending on the priority scheme in use.

The following delays are required between commands:

Read - Read — 160 nanoseconds
Read - Write — 500 nanoseconds
Write - Read — 500 nanoseconds
Write - Write — 190 nanoseconds

Example

This program sends end-of-int1 service at the end of the service interrupt routine:

```
ocw2          equ 20          ;port address of ocw2
command       equ 61          ;6 = seoi, 1 = init1

send_seoi:    mov al,command    ;set al=command
              mov dx,ocw2      ;set dx=port address
              out dx,al        ;send command
              sti              ;enable external interrupts
              ret              ;return to service routine.
```

Programmable Interval Timer

The INTEL 8254 Interval Timer has three identical, 16-bit, settable, decrementing counters. Each counter is totally independent. The counters have either a BCD or binary value and operate in one of five modes:

- **Interrupt on Terminal Count**

The output remains low after the mode is set. It continues low after the counter is loaded until the counter counts down to zero. Then the output goes high. It remains high until a new mode is selected or a new count is loaded.

- **Programmable One-Shot**

The output goes low one count following the rising edge of the gate input. The output goes high on the terminal count.

- **Rate Generator**

The output is low for one period of the input clock. The period from one output pulse to the next equals the number of input counts in the count register.

- **Square Wave Rate Generator**

The output remains low for one period of the input clock. The output remains high until one-half the count has elapsed. If the count is odd, the output is high for $(n+1)/2$ and low for $(n-1)/2$.

- **Soft Triggered Strobe**

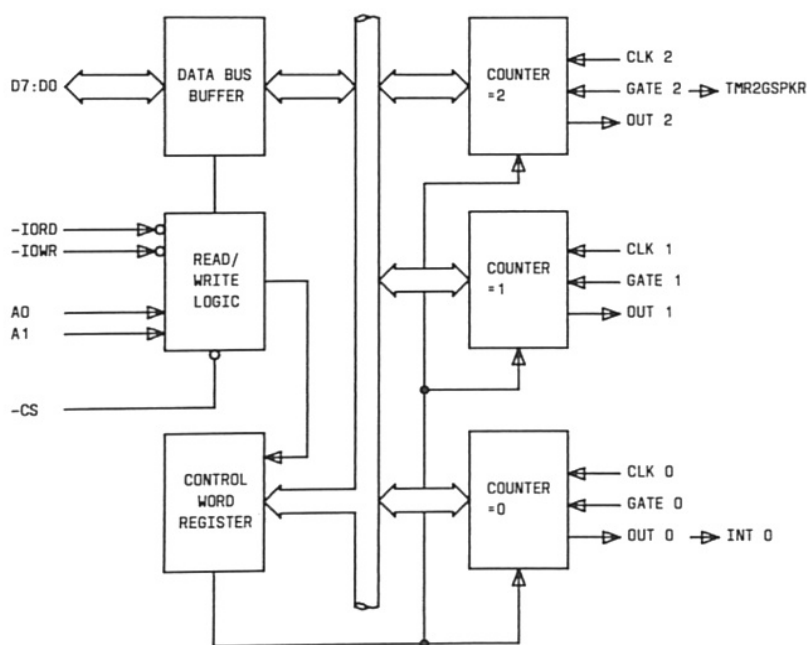
After the mode is set, the output will be high. When the count is loaded, the counter begins counting. On the terminal count, the output goes low for one clock period then goes high.

● Hardware Triggered Strobe

The counter starts counting after the rising edge of the trigger input and goes low for one clock period when terminal count is reached.

In the PC 6300 PLUS, Counter 0 provides real time interrupts on INT0; counter 1 requests memory refreshes; and counter 2 generates a pulse train for the audio speaker.

Block Diagram



FUNCTIONS

The Programmable Interval Timer can perform the following functions:

- Allows you to set a specific counter.
- Allows you to read a specific counter.

REGISTERS

PORT	NAME	NOTES
40	Counter 0	Provides real time interrupt INT0
41	Counter 1	Provides signals to refresh memory
42	Counter 2	Generates pulse train for the speaker
43	Timer Control	

REGISTER FORMATS

Timer Control Register (Write Only)

Bit	Contents
7	— Counter
6	
5	— Load
4	
3	— Mode
2	
1	
0	— Format

COUNTER

These bits indicate the counter as follows:

- 0 — Real-Time Clock (counter 0)
- 1 — DMA Refresh (counter 1)
- 2 — Tone Generator (counter 2)

LOAD

These bits indicate what part to load as follows:

- 0 — Read counter
- 1 — Load MSB only
- 2 — Load LSB only
- 3 — Load both MSB and LSB

Note: When loading both (3), the LSB will be loaded first.

MODE

These bits indicate the mode as follows:

- 0 — Interrupt on terminal count
- 1 — Programmable One-Shot
- 2 — Rate Generator
- 3 — Square Wave Rate Generator
- 4 — Software Triggered Strobe
- 5 — Hardware Triggered Strobe

FORMAT

This bit indicates the format as follows:

- 0 — Binary
- 1 — Binary Coded Decimal (BCD)

SEQUENCING AND TIMING

All counters must be initialized with the control register. The control register specifies the number of bytes which must be loaded.

Whenever a read or a load command is issued, the requested counter bytes must be read or written. In the read case, two reads are necessary, the first for the least significant byte (LSB) and the last for the most significant byte (MSB). In the write case, the control register specifies the byte to write.

One microsecond recovery time is required between a read or a load and any other control signal.

Input to the timer is 1.2288MHz. Therefore, there are 18.75 interrupts per second. To generate a 1.00-kHz tone with the audio speaker, a square wave rate generator is used with a count of 614 ($1.2288\text{MHz}/2 \times 614 = 1\text{-kHz}$).

Example

This program requests an interrupt in approximately 10 usec:

```

timer_control equ 43
timer0        equ 40

int_mask      equ 21

ask_for_intr:  mov al,0fh          ;allow only into
                                   ;interrupt
                out int_mask,al    ;send mask
                mov al,00110000b   ;binary counter,
                                   ;interrupt on terminal
                                   ;count, set count 0

                out time_control,al
                mov ax,12          ;12* 813.8 nanoseconds
                out timer0,al
                mov al,h          ;output counter, lsb
                                   ;then msb

                out timer0,al
                ret

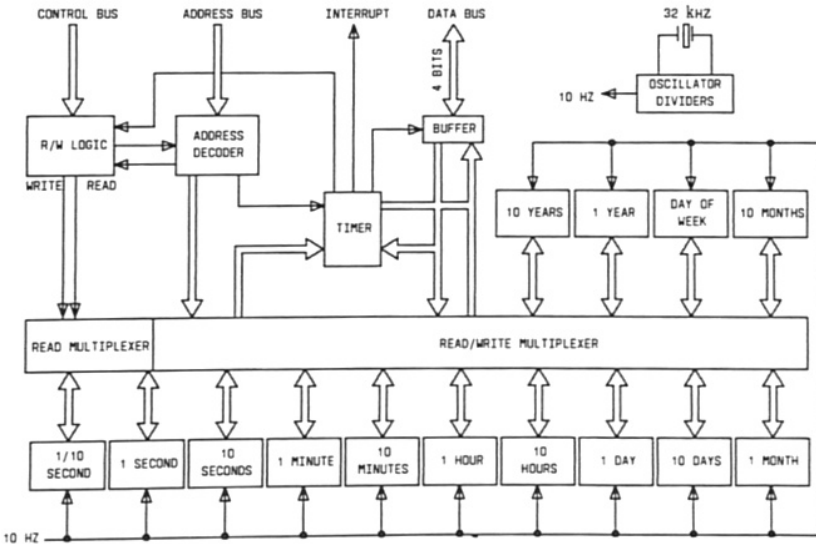
```

Real-Time Clock And Calendar

The real-time clock and calendar (MM 58274) keep the current date and time. All of the date and time fields can be read, but the second fields cannot be written.

The calendar keeps up to eight years. A rechargeable battery maintains operation even when the computer is off.

Block Diagram



FUNCTIONS

The real-time clock and calendar can perform the following functions:

- Read all calendar and time registers
- Write all calendar and time registers except seconds.

REGISTERS

PORT	NAME	NOTES
70	Control	0=Normal, 1=Test mode
71	Seconds — tenths (00:00:00.X)	X=0 through 9
72	Seconds — units (00:00:0X.0)	X=0 through 9
73	Seconds — tens (00:00:X0.0)	X=0 through 5
74	Minutes — units (00:0X:00.0)	X=0 through 9
75	Minutes — tens (00:X0:00.0)	X=0 through 5
76	Hours — units (0X:00:00.0)	X=0 through 9
77	Hours — tens (X0:00:00.0)	X=0 through 1
78	Days — units (00/0X/00)	X=0 through 9
79	Days — tens (00/X0/00)	X=0 through 3
7A	Months — units (0X/00/00)	X=0 through 9
7B	Months — tens (X0/00/00)	X=0 through 1
7C	Years — units (00/00/0X)	X=0 through 9
7D	Years — tens (00/00/X0)	X=0 through 9
7E	Day of the week	X=0 through 6 0 — Sunday 1 — Monday . . . 6 — Saturday
7F	Clock Setting / Interrupt Registers	

REGISTER FORMATS

Control Register (Read & Write)

Bit	Contents
7	
6	
5	— N/A
4	
3	— Test mode (Yes=1)
2	— Clock status (Run=0, Stop=1)
1	— Interrupt select
0	— Interrupt status

INTERRUPT SELECT

This bit indicates the interrupt selection as follows:

- 0 — Clock setting register
- 1 — Interrupt register

INTERRUPT STATUS

This bit indicates interrupt status as follows:

- 0 — Interrupt run
- 1 — Interrupt stop

Clock Setting Register (Read & Write)

Bit	Contents
7	— N/A
6	
5	
4	
3	— Leap year counter
2	
1	— AM/PM indicator
0	— 12/24-hour select

LEAP YEAR COUNTER

These bits indicate whether a year is a leap year as follows:

- 0 — Leap year
- 1 — Leap year +1
- 2 — Leap year +2
- 3 — Leap year +3

AMPM INDICATOR

This bit indicates AM/PM selection, in 12-hour mode, as follows:

- 0 — AM
- 1 — PM

12/24-HOUR SELECT

This bit indicates 12/24-hour selection as follows:

- 0 — 12-hour mode
- 1 — 24-hour mode

Interrupt Register (Read & Write)

Bit	Contents
7	
6	
5	— N/A
4	
3	— Type of interrupt
2	
1	— Duration
0	

TYPE OF INTERRUPT

This bit indicates the type of interrupt as follows:

- 0 — Single interrupt
- 1 — Repeated interrupt

DURATION

These bits indicate the duration of the interrupt as follows:

- 0 — No interrupt
- 1 — 0.1 seconds
- 2 — 0.5 seconds
- 3 — 1 second
- 4 — 5 seconds
- 5 — 10 seconds
- 6 — 30 seconds
- 7 — 60 seconds

SEQUENCING AND TIMING To write data to the clock and time registers, the unit must be out of test mode and stopped. After writing to the clock, it must be restarted.

To initialize interrupts, set Bit 4 in the Interrupt/Year mod 8 register. Write the register once and then read it in three times.

If an update occurs while reading a register, the illegal code of F is returned.

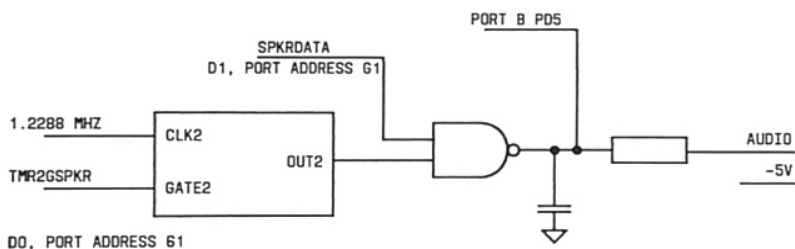
Example This program sets the time to 12 noon:

```
test_port      equ 70
stop_start     equ 7e
tens_hours     equ 77
write_time:    xor ax,ax           ;ax =
               mov dx,test_port    ;setup dx
               out dx,al           ;take out of test mode
               out stop_start,al    ;stop the clock
               mov dx,tens_hours    ;port adrs of 10's of hours
               mov al,1
               out dx,al           ;ten's of hours = 1
               dec dx
               mov al,2
               out dx,al           ;unit hours = 2
               dec dx
               xor al,al
               out dx,al           ;tens of minutes = 0
               dec dx
               out dx,al           ;minutes = 0
               mov al,0fh
               out stop_start,al    ;start the clock
               ret
```

Speaker

The speaker uses a permanent magnet speaker which is driven by one of two sources. Counter 2 of the Interval Timer can be programmed to automatically generate a pulse train. A bit in the Keyboard Control register controls this pulse train. A bit in the Keyboard Control register can also be programmed to manually generate a pulse train.

Block Diagram



REGISTERS

PORT	NAME	NOTES
61	Keyboard Control	See KEYBOARD INTERFACE
42	Counter 2	Generates pulse train for the speaker
43	Timer Control	See PROGRAMMABLE INTERVAL TIMER info

REGISTER FORMATS

To generate a sound set the parameters of the Keyboard Control and Timer Control registers as follows:

Keyboard Control Register (Write Only)

Bit	Contents
7	— 0
6	— 0
5	— 0
4	— 0
3	— 0
2	— 0
1	— 0
0	— Speaker

SPEAKER

This bit indicates whether the speaker is on or off as follows:

0 — Off

1 — On

Time Control Register (Write Only)

Bit	Contents
7	— 2
6	— 3
5	— 3
4	— 3
3	— 3
2	— 3
1	— 3
0	— Format

FORMAT

This bit indicates the format as follows:

0 — Binary

1 — Binary Coded Decimal (BCD)

SEQUENCING AND TIMING Input to the timer is 1.2288MHz. To generate a 1.00-kHz tone with the audio speaker, a square wave rate generator is used with a count of 614 ($1.2288\text{MHz}/2*614 = 1\text{-kHz}$).

Example This program sounds the speaker:

```
key_control    equ 61
beep:          mov dx, key_control    ;port address of Keyboard
              in al,dx
              mov ah,al              ;save control register
              mov bl,80h             ;outer counter
beep1:         and al,0fch           ;turn off speaker and
              out dx,al              ; manual pulse bit
              mov cx,48h
              loop $
              or al,02h              ;turn on manual pulse
              out dx,al              ; bit
              mov cx, 48h
              loop $
              dec bl
              jnz beep1
              mov al,ah
              out dx,al              ;restore keyboard control
              ret                    ; register
```

Switching Between Real and Protected Modes

The following program is from the power up routine in the ROM BIOS. At this point in the power up routine, the data segment is 40H and the stack segment is 30H.

Note: Do not use 30H for your stack segment.

The variables **real_loc1** and **real_loc2** are defined to be double words at locations 40:A2 and 40:A6. Location 40:A2H is used for the real mode return addresses.

Also, it is necessary for the power up routine to copy the **GDT** entries from ROM to RAM since the 80286 microprocessor dynamically updates the **GDT** as well as this program. The temporary power-up **GDT** is defined to begin at location 40:AA.

Once in protected mode, the segment registers **MUST** be loaded with valid **GDT** selectors if they are changed. However, they can use the real mode value that they contained when you entered protected mode. Upon reset CS=F000H, MSW=FFF0H, flags=0002H, IP= FFF0H, DS=SS=ES=0000H.

Note that while in protected mode this code makes **NO** attempt to field or service interrupts (no **IDT** is setup). It also makes no attempt to switch tasks (no **LDT** or **TSS** registers are initialized). This code simply shows how to get in and out of protected mode and access memory above 1M byte.

```

code segment public 'ROM'
    assume cs:code, ds:nothing, es:nothing, ss:nothing
    assume cs:code, ds:data, es:abs0, ss:stack_ram

    data_seg      EQU      40H
    stack_seg     EQU      30H

; jump into protected mode and test memory above 640K bytes%

pmemcnt:
    assume cs:code, ds:data
    mov     ax,data_seg
    mov     ds,ax                ;satisfy assumptions
    call    i_gdt                ;initialize standard
                                ;gdt entries
    mov     ax,word ptr cs:[offset ptestaddr] ;initialize
                                ; gdt code segment%
    mov     word ptr ds:[gdt+8+2],ax

    mov     ax,cs:[offset wereback]
    mov     word ptr [real_locl],ax ;return location
                                ; in memory
    mov     word ptr ds:[real_locl+2],cs ;%
;   pusha                                ;save the world%
    db      60H
    push    ds
    push    es
    cli                                ;stop all interrupts%
    mov     bx,ds:[offset gdtalias]
;   lgdt    [bx]
    db      0FH,01H,17H

    mov     dx,3F20H                ;FF port%
    mov     al,90H                  ;enable upper 4 data lines%
                                ;return to wereback upon reset
    out     dx,al                    ;set the FF
;   smsw    ax                        ;machine status word into ax
    db      0FH,01H,0E0H
    or      ax,1                    ;set Protection Enable bit
;   lmsw    ax                        ;enable protection
    db      0FH,01H,0FOH
    jmp     foo                      ;clear prefetch que

foo:
    jmp     dword ptr cs:[holdon] ;indirect jump to 8:0
holdon:
    dw      0                        ;fully into protected mode
    dw      8

```

```

wereback:
    mov     ax,stack_seg
    mov     ss,ax                ;restore stack segment
    pop     es                  ;restore yourself%
    pop     ds
;   popa
    db      61H
    mov     dx,3F20H            ;FF port%
    mov     al,0H               ;so CTRL-ALT-DEL works properly
    out     dx,al               ;set the FF
    sti                      ;enable interrupts%

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;ptestaddr is the code that runs in protected mode to access
;memory above 1M byte. In this case, it reads location 0FB0000H
;and stores the complement of the contents in a low memory
;location to be accessed from real mode.
;
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
ptestaddr:
    mov     ax,10H
    mov     es,ax                ;set up es selector%
    xor     di,di                ;clear di%
    mov     ax,word ptr es:[di]  ;mov es:[di] into ax%
    not     ax
    mov     ds:[si]:ax           ;ds can still be used in
                                ;protected mode with its real
                                ;mode value just don't try to
                                ;load it with anything other
                                ;than a valid GDT SELECTOR
    mov     al,0                ;no error

history:
                                ;in location real_loc2+2%
    mov     byte ptr ds:[real_loc2+2],al
                                ;error code to report%
    mov     dx,3F00H            ;reset port%
    in      ax,dx                ;bang the port%
                                ;back to real mode now
    ;
    nop                                ;incase prefetch que%
    nop                                ;has fetched ahead%
    nop
    cli                                ;should never get%
    hlt                                ;to this point%
    ;

```

```

;-----
;      Install GDT ENTRIES
;
;      Input:  None:
;      Output: None.
;
;      Trash: ax=cx=0 destroyed.
;-----

i_gdtproc      near
                assume cs:code,ds:nothing,es:nothing,ss:stack_rm

                push    ds                ;save registers
                push    es
                push    di
                push    si

;Initialize the gdt with gdt_ent.

                assume  cs:code,ds:code,es:data,ss:stack_rm

                mov     ax,cs              ;satisfy assumptions
                mov     ds,ax              ;
                mov     ax,data_seg
                mov     es,ax              ;es:di=data_seg:gdt
                mov     ax,es:[offset gdt]
                mov     di,ax
                mov     ax,cs:(offset gdt_ent) ;ax=offset gdt_ent
                mov     si,ax              ;si=offset gdt_ent
                mov     cx,16H             ;loop 16 times for 16 words

i_gdt0: movsw                ;es:di++ gets ds:si
        loop    i_gdt0        ;until cx=0.

        pop     si              ;restore registers
        pop     di
        pop     es
        pop     ds
        ret

gdt_ent        dw     0,0,0,0          ;first entry all 0's
                dw     0FFFFH          ;next entry 8H
                dw     0                ;to be filled in
                db     0FH              ;segment 0F000H
                db     9AH              ;kernel code protection
                dw     0                ;always 0

```

```

        dw  0FFFFH      ;next entry 10H
        dw  0           ;offset is 0
        db  0FBH        ;want to access phys adrs
                        ;FBxxxxH
        db  92H         ;kernel data
        dw  0           ;always 0
gdtalias  dw  32         ;limit of gdt
        dw  4AAH        ;offset to gdt
        db  0           ;segment of gdt
        db  92H         ;kdata
        dw  0           ;always 0
i_gdt     endp

code ends

```

The GDT can be visualized as follows beginning at 4AAH:

0	00	00
	00	00
	00	00
	00	00
8h	FF	FF
	PTESTADDR	
	0F	9A
	00	00
10h	FF	FF
	00	00
	FB	92
	00	00
gdtalias 18h	20	00
	AA	04
	00	92
	00	00

THIS ENTRY ALWAYS ALL 0

LIMIT IS FFFF
ADDRESS OF PTESTADDR
HI PART PTESTADDR IS 0FH ACCESS RIGHTS
ALWAYS 0

LIMIT IS FFFF
BASE IS 0
PHYS ADDR FB0000 IS USED ACCESS RIGHTS
ALWAYS 0

LIMIT OF GDT
PHYSICAL ADDR OF GDT
HI PART OF GDT PHYS ADDR ACCESS RIGHTS
ALWAYS 0



DEB Capabilities	10-3
The DEB Driver	10-4
16-Color Graphics	10-5
Look-Up Table (LUT)	10-6
Overlay Modes	10-6
Programming Tips	10-8
Hardware/Software Compatibility	10-8
Setup	10-9
How To Program The DEB	10-10
Mode Setting	10-10
Setting Colors and Effects	10-12
Displaying Graphics Images	10-12
Interrupt 10H Functions	10-13
Set Display Mode	10-14
Set Cursor Position	10-15
Read Cursor Position	10-16
Select Active Display Page	10-16
Scroll Active Page Up	10-17
Scroll Active Page Down	10-18
Read Character and Attribute at Cursor Position	10-19

Write Character and Attribute at Cursor Position	10-20
Write Character Only at Cursor Position	10-20
Set Color Palette	10-21
Write Dot	10-28
Read Dot	10-28
Write Teletype to Active Page	10-29
Read Current Video State	10-29
Programming the LUT	10-30
16-Color Graphics LUT Programming	10-31
Overlay Modes—LUT Programming	10-46
Programming the Bit Planes	10-56

DEB Capabilities

The Display Enhancement Board (DEB) option adds improved color and graphics to your AT&T Personal Computer 6300 PLUS.

When you use the DEB with the PC 6300 PLUS color monitor, you can display graphics in up to 16 color combinations simultaneously or treat the screen as two screens in one and overlay one screen treatment on top of the other.

When you use the DEB with the PC 6300 PLUS monochrome monitor, you have the same capabilities you have with the color monitor, except that colors are displayed as "shades of green."

The DEB is compatible with existing software, so all the programs you have already can be used now as if the DEB were not installed. Of course, these programs may not have access to any of the new capabilities.

This chapter describes the functionality of the DEB device driver. Although it is not necessary to use the driver in order to use the DEB, the driver is designed to work with MS-DOS, GWBASIC, and other AT&T software products. To program the DEB hardware directly, refer to the *AT&T Technical Reference Manual*. However, such programming is considered circumvention of the AT&T operating system and we advise against it.

This chapter assumes that you are familiar with video programming through the Interrupt 10H interface and with assembler programming. Information on the Interrupt 10H interface can be found in Chapter 8, **BIOS Routines**.

Before you begin writing programs for the DEB, follow the procedures in the DEB Installation Manual for installing the DEB hardware and device driver software.

The DEB is an optional hardware component for the PC 6300 PLUS that works in conjunction with the built-in Video Display Controller (VDC) to provide improved color and graphics.

The built-in VDC contains circuitry and memory that support either 4 color medium resolution (320×200 pixels) graphics, 1 color high resolution (640×200 pixels) graphics, or 1 color super resolution (640×400 pixels) graphics.

The DEB contains additional circuitry and memory that can be combined with the capabilities of the built-in VDC to produce up to 16 color combinations in either high or super resolution. You can also program the VDC and DEB separately, treating them as two separate images that are combined on one screen to produce an overlaying effect. The overlay modes let you use up to 8 colors on the DEB screen and up to 16 colors on the VDC screen.

THE DEB DRIVER

You load the DEB device driver by entering a "DEVICE" statement in the CONFIG.SYS file (see **Programming Tips** heading.) The driver installs an Interrupt 10H "filter" during the loading process.

When you are using the DEB and are running some programs that use the DEB and some that do not, the "filter" provides video support for both kinds of programs. For programs that do not use the DEB, the filter passes control to the standard Interrupt 10H ROM BIOS routine.

The DEB driver installs a filter for Interrupt 9H. This filter resets the DEB to transparent mode whenever you warmstart the system through **CTRL ALT DEL**. The filter controls scrolling when you press **CTRL NUMLOCK**.

16-COLOR GRAPHICS

This feature lets you display 16 color combinations in either high resolution (640×200) or super resolution (640×400). Not only can you use the standard 16 colors, you can also combine colors to form new colors and cause pixels to blink from one color to another.

The DEB provides 5 palettes for you to use when programming in color. At any point in your program, you select one of the palettes as the "active" palette. The color combinations contained in that palette determine what colors and effects show on the screen.

Each of the first 4 palettes contains a default set of 16 color combinations, but to suit the needs of your program you can change the contents of the palette to any one of the following:

- Any of the 16 standard colors with which you are already familiar from the standard applications. The standard colors are:

0 = black	8 = gray
1 = blue	9 = light blue
2 = green	10 = light green
3 = cyan	11 = light cyan
4 = red	12 = light red
5 = magenta	13 = light magenta
6 = brown	14 = yellow
7 = white	15 = high intensity white

- A mixture, or "dithering," of any 2 of the 16 standard colors
- An alternation, or blinking, between any 2 of the standard 16 colors.

The last palette contains no default combinations. You program the fifth palette by loading color values into a 256-byte array. The DEB device driver uses this special palette to program the DEB's color Look-Up Table (LUT). By using the LUT you can add the capability of dithering or blinking between any four colors.

LOOK-UP TABLE (LUT)

The LUT resides in RAM on the DEB board, and is accessed through write-only hardware registers. The device driver keeps a copy of the register values in the LUT. The register values are accessible to software applications through the device driver. The LUT contains 256 values that determine the colors, blinking, and dithering that appear on the screen. Whether you need to learn about the use and layout of the LUT depends on the application you are writing.

When using the standard palettes, you need not be concerned with the LUT. The DEB device driver automatically programs the LUT to correspond to the way you set up the palettes. When you program a custom LUT, you greatly increase the color combinations and blinking effects available to you.

OVERLAY MODES

The overlay modes let you use the screen to display two images at once, independently. For example, you can display a high resolution color graphics image with its own foreground and background. Then, on "top" of that image, you can display a box of text and scroll the text without affecting the graphics image.

The overlay modes use the DEB to control one image and use the standard controller board to control the other image. You can select from many combinations of graphics, text, color, and high or super resolution in designing the two images.

The overlay modes offer 5 palettes. Each of the first 4 palettes has 8 positions. These four palettes have default colors that you can change to suit your needs. You can choose 8 color combinations from any of the 16 standard colors, or blink between 2 of the standard colors. The dithering combinations of the 16-color graphics modes are not available. You can also use the 5th palette to custom program the LUT.

Programming Tips

Whenever you plan an application, it is important to use the DEB device driver to test for the presence of both the DEB and the associated driver. Test for the presence of the hardware by checking for DEB video memory. You can do this by writing data patterns into memory, in the range A000H:0H to B800H:0H, and then reading them back. Test for the software device driver by issuing a function call to open the device called "DEBDRIVE," then immediately issuing a call to close "DEBDRIVE." If the open fails (carry set on return from Interrupt 21H) the drive is not present. No functions are implemented in the driver, which is used only to detect the presence of the software.

HARDWARE/ SOFTWARE

COMPATIBILITY

The driver software has been designed to fit into the structure of MS-DOS programs. The DEB hardware uses the same range of addresses as the standard video ports on any compatible machine. If your application uses a light pen, consult the DEB supplement in the *AT&T Personal Computer Technical Reference Guide*.

The DEB driver makes minor modifications to the ROM BIOS video interrupt. Mode setting and color selection offer additional functionality. Be careful when you use the following functions.

- SET MODE—uses an additional register BL
- SCROLLING—uses an additional register BH

- STATUS—returns an additional register pair ES:DI. No application should count on ES:DI not changing.

SETUP

Install the DEB driver just as you would install any device driver. Be sure the CONFIG.SYS file is in the root directory. Put the line

DEVICE = DEDRIVER.DEV

in CONFIG.SYS. This line puts the DEB driver in the device driver chain. The driver makes patches in INT 10H and INT 9H to add the new functionality. The driver has two features:

- The INIT function, which deallocates itself after it runs
- Chaining, which allows you to test for the driver's presence by issuing an open function call.

How To Program The DEB

There are three steps for video programming that apply whether or not you are using the DEB capability:

- 1 Set the hardware's mode. You also must set the active page if you are in an overlay mode and want to select the DEB screen.
- 2 Select the color combinations and effects you want to use.
- 3 Construct the graphics images you want to display.

To program the LUT see **Programming the LUT** heading.

MODE SETTING

The DEB is controlled by invoking one of the DEB video modes through the Set Mode function (INT 10H, function 0H). The Set Mode function establishes the mode for both the DEB and the VDC.

These modes fall into four categories:

- 16-Color Graphics
- Overlay
- Transparent
- Disabled

**16-Color
Graphics
Modes**

There are two DEB modes that provide 16-color graphics high resolution and super resolution. Both these modes let you use 5 palettes and display up to 16 color combinations simultaneously.

**Overlay
Modes**

When overlaying the VDC on the DEB output, you specify one of the modes for the VDC and one mode for the DEB. The VDC modes are a subset of the modes for non-DEB graphics: 80 × 25 text mode, high and super resolution modes. The DEB modes are both graphics modes: high and super resolution.

When using one of the four standard palettes, the VDC's output takes precedence over the output of the DEB, so that if each board writes a pixel to the same screen location, the pixel sent by the VDC is displayed. This precedence is programmed into the LUT. To have the DEB take precedence over the VDC, you must change the values in the LUT. (For more information, see **Programming the LUT** heading.)

**Transparent
Mode**

The non-DEB modes, modes 0-40H and mode 48H, work exactly as they work without the DEB device driver installed.

**Disabled
Mode**

In the disabled mode, you can cause the output of the VDC, the DEB, or both to be blacked out. This enables you to draw a graphics image or to fill a screen with text and not have them displayed while you are building them. You can then have the image "pop up" by taking VDC or DEB out of the disabled mode.

You can also achieve this result by using the programmable palettes and the LUT.

SETTING COLORS AND EFFECTS

Colors and effects are controlled by the Set Color Palette Command, (INT 10H function 0BH). Use this function to set color values in one of the four palettes, to switch between palettes, or to reset palettes to their default values. You also use Set Color Palette to program the LUT directly.

DISPLAYING GRAPHICS IMAGES

There are two methods for displaying graphics images using the DEB: writing dots at screen locations or directly programming the VDC and DEB memory.

To write dots (pixels) to the screen, use the Write Dot function (INT 10H, function 0CH). Write Dot requires that you specify the display page, the row and column where you want the dot to appear, and the color or pattern for the dot.

To program the VDC and DEB graphics memory directly, you need to learn the details of how the LUT is structured and how LUT addresses are formed (see **Programming the LUT** heading.)

Interrupt 10H Functions

This part describes the DEB device driver software functions. This interface is an extension of the INT 10H software to the PC 6300 PLUS ROM BIOS that controls the VDC. The ROM BIOS screen driver has the following functions:

Code	Functions
0H	Set the display mode
2H	Set the cursor position
3H	Read the cursor position
5H	Select the active display page
6H	Scroll the active page up
7H	Scroll the active page down
8H	Read character and attribute at cursor position
9H	Write character and attribute at cursor position
AH	Write only the character at cursor position
BH	Set the color palette
CH	Write a dot on the screen
DH	Read a dot on the screen
EH	Write in teletype style to the active page
FH	Return information about current video state

Not all of these functions are applicable to the DEB. The filter receives the Interrupt 10H function call, filters the functions that are applicable to the DEB and performs them. The functions that are not applicable to the DEB are passed on to the ROM BIOS INT 10H routine or to a previously installed filter or driver routine. The following section describes the functions which are processed by the DEB Interrupt 10H filter.

SET DISPLAY MODE

This function establishes the mode for both the DEB and the VDC. If you select a non-DEB related mode, control is passed to the ROM resident Set Mode function. Set Mode initializes palette 0 as the active palette.

Input

AH = 0H function number for Set Mode

AL = new mode

BL = optional overlay mode

Setting AL bit 7 = 0 puts you in either the DEB transparent mode or 16-color graphics mode:

AL = 0H — 40 X 25 monochrome, text

AL = 1H — 40 X 25 color, text

AL = 2H — 80 X 25 monochrome, text

AL = 3H — 80 X 25 color, text

AL = 4H — 320 X 200 color

AL = 5H — 320 X 200 monochrome

AL = 6H — 640 X 200 color

AL = 40H — 640 X 400 with 2-position program-
mable palette, defaulting to black
and white

AL = 41H — 640 X 200 16-color graphics with
four palettes

AL = 42H — 640 X 400 16-color graphics with
four palettes

AL = 44H — Disable mode (disables both DEB
and VDC output)

Setting AL bit 7 = 1 puts you in overlay mode. The following values are only used in overlay mode. AL contains the setting for the VDC; BL contains the mode setting for the DEB. In overlay modes, the active page defaults to zero.

VDC Settings

AL = 82H — 80 X 25 monochrome, text
 AL = 83H — 80 X 25 color, text
 AL = 86H — 640 X 200 color graphics
 AL = 0C0H — 640 X 400 color graphics
 AL = 0C4H — Disable mode. Disables only VDC

DEB Settings

BL = 6H — 640 X 200 graphics with four 8-position palettes.
 BL = 40H — 640 X 400 graphics with four 8-position palettes.
 BL = 44H — Disable mode. Disables only DEB

Output Contents of all registers are preserved.

Example

MOVAH,0 ;Select Set Mode
 MOVAL,41H ;Select 16 color graphics
 INT 10H ;Change the mode

SET CURSOR POSITION

This function sets the cursor position for either the DEB, the VDC, or both.

Input

AH = 2H — Function number for Set Cursor Position
 DH,DUC — Row, column of new position
 BH — Page number

Valid page numbers for DEB modes are 0 for the VDC and 80H for the DEB in the overlay mode. Row values are 0 thru 23, column values are 0 thru 79, in DEB modes.

Output Contents of all registers are preserved.

Example

MOVAH,2 ;SCP function
 MOVDH,ROW
 MOVDL,COL
 MOV BH,PAGE
 INT10H ;Moves cursor to position defined in above variables.

READ CURSOR POSITION

This function returns the position of the cursor for the DEB, VDC, or both.

Input

AH = 3H — Function number for Read Cursor
Position
BH — Page number

Valid page numbers for DEB modes are 0 for the VDC and 80H for the DEB in overlay mode. Row values are 0 thru 23, column values are 0 thru 79, in DEB modes.

Output

DH,DL — Row (DH), column (DL) of current position. Contents of all other registers are preserved.

Example

```
MOVAH,3  
MOVBH,PAGE  
INT 10H  
MOVROW,DH  
MOVCOL,DL
```

SELECT ACTIVE DISPLAY PAGE

This command enables the selection of the DEB display in the overlay mode.

Input

AH = 5H — Function number for Select Active
Display Page
AL — Active page

Values for the active page in DEB modes are: 0 for the VDC and 80H for the DEB.

Output

Contents of all registers are preserved.

Example

```
MOVAH,5  
MOVAL,PAGE  
INT 10H
```


**SCROLL
ACTIVE
PAGE UP**

This function defines a pattern that is to be displayed on the blank lines as the screen scrolls. The pattern consists of ones and zeros. Zeros are interpreted as the background color (palette position zero). Ones are interpreted as the foreground color, which is defined in BL.

Care should be taken when scrolling in DEB modes to insure that all applications set the additional argument in BH correctly.

Input

AH = 6H — Function number for Scroll Active Page Up

AL — Number of lines to scroll

CH,CL — Row (CH), column (CL) of upper left corner to scroll

DH,DL — Row, column of lower right corner to scroll

BH — Pattern to be used on blank lines

BL — Foreground color

The range of lines to be scrolled is 0 thru 23 (where 0 specifies clear screen).

Row values are 0 thru 23, column values are 0 thru 79, in DEB modes.

Valid foreground colors are specified by palette positions 0 thru FH for 16-color graphics, and 0 thru 7H for 8-color graphics.

Output

Contents of all registers are preserved.

Example

```
MOVAH,6           ;Scroll Active Page Up
MOVAL,LINES
MOVCH,UPROW
MOVCL,UPCOL
MOVDH,LOWROW
MOVDL,LOWCOL
MOVBH,0
MOVBL,FGCOLOR
INT 10H
```

SCROLL ACTIVE PAGE DOWN

This function permits you to define a pattern that is to be displayed on the blank lines as the screen scrolls downward. The pattern consists of ones and zeros. Zeros are interpreted as the background color (palette position zero). Ones are interpreted as the foreground color, which is defined in BL.

Care should be taken when scrolling in DEB modes to insure that all applications set the additional argument in BH correctly.

Input

AH = 7H — Function number for Scroll Active Page Down

AL — Number of lines to scroll

CH,CL — Row (CH), column (CL) of upper left corner to scroll

DH,DL — Row (DH), column (DL) of lower right corner to scroll

BH — Pattern to be used on blank lines

BL — Foreground color

The range of lines to be scrolled is 0 thru 23 (where 0 specifies clear screen).

Row values are 0 thru 23, column values are 0 thru 79, in DEB modes.

Valid foreground colors are specified by palette positions 0 thru FH for 16-color graphics, and 0 thru 7H for 8-color graphics.

Output

Contents of all registers are preserved.

Example

```
MOVAH,7      ;Scroll Active Page Down
MOVAL,LINES
MOVCH,UPCOL
MOVDH,LOWROW
MOVDL,LOWCOL
MOVBH,0
MOVBL,FGCOLOR
INT 10H
```

**READ
CHARACTER
AND
ATTRIBUTE
AT CURSOR
POSITION**

This function returns the value of the character at the current cursor position. The value of the character's foreground color is returned in AH.

Input

AH = 8H — Function number for Read Character and Attribute at Cursor Position.

BH — Page

Valid page numbers for DEB modes are 0 for the VDC and 80H for the DEB in overlay mode.

Output

AL — ASCII character code

AH — Foreground palette position or VDC attribute

Contents of all other registers are preserved.

Example

```
MOV AH,8      ;Read CHR function
MOV BH,PAGE
INT 10H
MOV CHAR,AL    ;save CHAR/COLOR
MOV CURCOLOR,AH
```

**WRITE
CHARACTER
AND
ATTRIBUTE
AT CURSOR
POSITION**

This function displays the character whose ASCII code is in register AL. The character is displayed according to the color values in BL.

Input

AH = 9H — Write Character function
AL — ASCII character code
BL — Foreground color
BH — Page
CX — Count of characters to write

If bit 7 of BL = 1, the color value is XOR'd with the current dots in that location.

Valid page numbers for DEB modes are 0 for the VDC and 80H for the DEB in overlay mode.

Valid foreground colors are specified by palette position 0 thru FH for 16-color graphics, and 0 thru 7H for 8-color graphics.

Output

Contents of all registers are preserved.

Example

```
MOV AH,9
MOV AL,CHAR
MOV BL,CURCOLOR
MOV CX,1
INT 10H
```

**WRITE
CHARACTER
ONLY AT
CURSOR
POSITION**

In DEB modes, this function is the same as "Write Character and Attribute."

**SET COLOR
PALETTE**

This function is used to set color values in one of the four palettes, to switch between palettes, or to reset palettes to their default values.

In the overlay modes, the Set Color Palette function works on the active page. If the active page is set to display to the VDC board, this function works the same as the standard ROM BIOS INT 10H (function BH).

If you specify a palette position greater than the value allowed for the mode in which you are working, the value you specify will be put in that palette's highest position. For example, if you attempted to set palette position 13 to red when working in overlay mode, which has 8-position palettes, the 8th palette position would be set to red.

Note: The following discussion covers the use of the simple palette programming functions. You can also use "Set Color Palette" to program the LUT. (For more information, see **Programming the LUT** heading.)

Input

AH= 0BH — Function number for Set Color Palette

AL — Palette function selector

BH — Positional pointer

BL — Color value

For simple palette programming functions, use the following:

AL = 0

BH = palette color ID

BH = FFH — Switches to the palette specified in BL, without changing to the default palettes unless there is a change in palette type (e.g., change from a 16-position palette to an 8-position palette).

BH = 80H — Switches to the palette specified in BL and resets the palette to its default.

BH = — 0-16 sets this palette position to the color or attribute in BL.

BL — Actual color value or code for blinking and dithering

Comments

The DEB driver lets you automatically load your customized LUT and use it in place of one of the standard palettes.

The steps for loading and using the customized LUT are:

- 1 Define the table with DB (Define Byte) statements.
- 2 Load the table in by using the Set Color Palette command.
- 3 Use the Read Dot and Write Dot functions to access the LUT (see **Interrupt 10H Function** heading).

The code for defining the table would be similar to this:

```
LUT-STRING DB 4      !Signifies active palette 4
             DB 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
             DB 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
             DB 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
             DB 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
             .
             .
             .
             DB 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
```

Customized LUT Input

The special settings for using a customized LUT in Set Color Palette are as follows:

AL = non-zero (a zero here selects a standard palette)

AL bit 0 = 1 means use ES:SI to program the palette and registers BH and BL to indicate an offset and length into the LUT. ES:SI points to the LUT table (in the above example, LUT-STRING).

In this case, BH = offset into LUT and BL = length of portion of LUT to be changed. When loading an entire new table, set BH and BL to 0.

AL bit 1 = 1 means use BH and BL to program the LUT one location at a time.

In this case, BH = position in LUT and BL = the value to put in that position.

AL bit 2 = 1 means use the short LUT addressing mode. (Only uses the first 16 LUT entries.)

To load a new table of values into the LUT, where the table in your program is named LUT_STRING, you can use these statements:

```
PUSH DS    !save the data segment address
POP ES
MOV SI,LUT-STRING
MOV AL,1
MOV AH,11
XOR BH,BX  !Sets BH=BL=0
INT 10
```

Default Palettes

The defaults for each of the four palettes are:

Palette Number 0

Position	Color
0	0 = black
1	2 = green
2	4 = red
3	6 = brown
4	1 = blue
5	3 = cyan
6	5 = magenta
7	7 = white
8	8 = gray
9	9 = light blue
10	10 = light green
11	11 = light cyan
12	12 = light red
13	13 = light magenta
14	14 = yellow
15	15 = high-intensity white

Palette Number 1

Position	Color
0	0 = black
1	3 = cyan
2	5 = magenta
3	7 = white
4	1 = blue
5	2 = green
6	4 = red
7	6 = brown
8	8 = gray
9	9 = light blue
10	10 = light green
11	11 = light cyan
12	12 = light red
13	13 = light magenta
14	14 = yellow
15	15 = high-intensity white

Palettes 2 and 3 are the same, and they contain the standard colors in numerical order.

Palette Numbers 2 and 3

Position	Color
0	0 = black
1	1 = blue
2	2 = green
3	3 = cyan
4	4 = red
5	5 = magenta
6	6 = brown
7	7 = white
8	8 = gray
9	9 = light blue
10	10 = light green
11	11 = light cyan
12	12 = light red
13	13 = light magenta
14	14 = yellow
15	15 = high-intensity white

**Blinking
Color Effects
for DEB
Palettes 0-3**

Color combinations 16-135 have been pre-assigned to allow you easy access to blinking effects while using the standard palettes. The following table describes the available combinations.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
2		31	32	33	34	35	36	37	38	39	40	41	42	43	44
3			45	46	47	48	49	50	51	52	53	54	55	56	57
4				58	59	60	61	62	63	64	65	66	67	68	69
5					70	71	72	73	74	75	76	77	78	79	80
6						81	82	83	84	85	86	87	88	89	90
7							91	92	93	94	95	96	97	98	99
8								100	101	102	103	104	105	106	107
9									108	109	110	111	112	113	114
10										115	116	117	118	119	120
11											121	122	123	124	125
12												126	127	128	129
13													130	131	132
14														133	134
15															135

Note: To select a blinking value, find the intersection of the two color codes. The color codes correspond to the codes found in the table under **Palette Number 2 And 3**.

**Dither
Combinations
for DEB
Palettes 0-3**

Color combinations 136-255 have been preassigned to allow you easy access to dithering effects while using the standard palettes. The following table describes the available combinations.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	136														
2	137	138													
3	139	140	141												
4	142	143	144	145											
5	146	147	148	149	150										
6	151	152	153	154	155	156									
7	157	158	159	160	161	162	163								
8	164	165	166	167	168	169	170	171							
9	172	173	174	175	176	177	178	179	180						
10	181	182	183	184	185	186	187	188	189	190					
11	191	192	193	194	195	196	197	198	199	200	201				
12	202	203	204	205	206	207	208	209	210	211	212	213			
13	214	215	216	217	218	219	220	221	222	223	224	225	226		
14	227	228	229	230	231	232	233	234	235	236	237	238	239	240	
15	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Note: To select a dithering value, find the intersection of the two color codes. The color codes correspond to the codes found in the table under **Palette Number 2 And 3**.

WRITE DOT This function writes a pixel to the Dot location on the screen that you specify. If the screen is in the DEB mode, Write Dot may also write a pattern.

Input AH = 0CH — Function number for Write Dot
 AL — Palette position to be written.
 BH — Display page designator (bit 7=1
 selects the DEB)
 CX — Column number
 DX — Row number

Output Contents of all registers are preserved.

Example MOV AH,0CH
 MOV AL,PALPOS
 MOV BH,PAGE
 MOV CX,COL
 MOV DX,ROW
 INT ;Write the Dot

READ DOT This function reads a dot from the screen. If the screen is in the DEB mode this function returns the value in the LUT that corresponds to this dot. (For more information, see **Programming the LUT** heading.)

Input AH = 0DH — Function number for Read Dot
 BH — Display page designator (bit 7=1
 selects the DEB)
 CX — Column number
 DX — Row number

Output AH — VDC value or DEB palette position.
 Contents of all other registers are preserved.

Example MOV AH,0DH
 MOV BH,PAGE
 MOV CX,COL
 MOV DX,ROW
 INT
 MOV DOTCOL,AH ;Save the Dot

**WRITE
TELETYPE
TO ACTIVE
PAGE**

This function works the same way whether you are writing to the DEB or to the VDC screen.

Input

AH = 0EH — Function number for Write Teletype
 AL — Character to write
 BL — Foreground color (in graphics modes)
 If bit 7=1, color is XOR'd t current contents.

Output

Contents of all registers are preserved.

Example

```
MOV AH,0EH
MOV AL,CHAR
MOV BL,FGCOL
INT
```

**READ
CURRENT
VIDEO
STATE**

This function returns the current video state. It indicates whether the DEB or VDC is active in the overlay mode and returns the number of the active palette.

Input

AH = 0FH — Function number for Read Current Video State
 BH — Display page designator
 AL — Mode currently set
 ES:DI — Pointer to a copy of the current LUT

Example

```
MOV AH,0FH
INT 10H
```

Programming the LUT

By programming the LUT yourself, you can create color patterns that are not available when you use standard palettes. You need not read this chapter if you do not want to use this extended functionality.

The hardware uses the LUT to translate the contents of video memory patterns into graphics effects. In the standard palettes, INT 10H filter programs the LUT for you and thereby provides the preassigned color combinations and effects as described in previous sections.

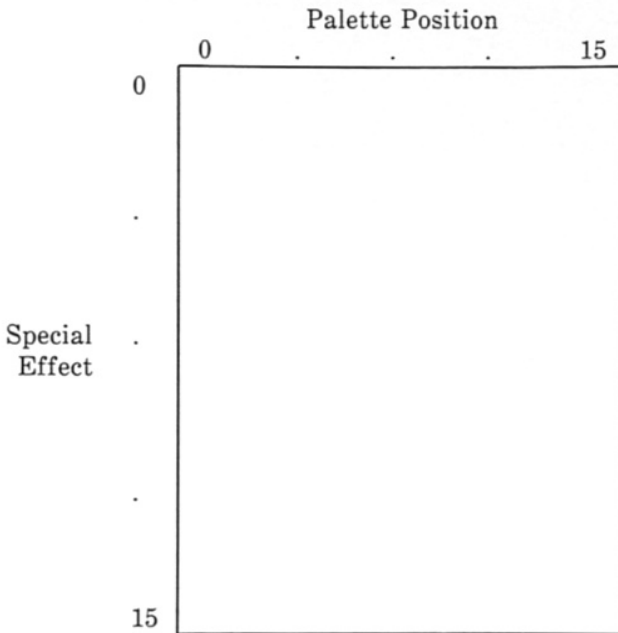
To program the LUT directly, you select Palette 4 in Set Color Palette function. Palette 4, also called the "LUT palette," has a minimum of 256 positions. Each palette position contains a value between 0 and 15. These values map into the LUT locations on the DEB. The 256 locations on the DEB collectively determine the color and special effects displayed when you specify a particular palette position. The color and special effect for each pixel on the screen are determined by:

- The palette position you specify
- The values in the LUT
- The active mode.

There are some differences in the way the LUT is structured for 16-color graphics modes and overlay modes. This part describes LUT operation for 16-color graphics modes and overlay modes separately.

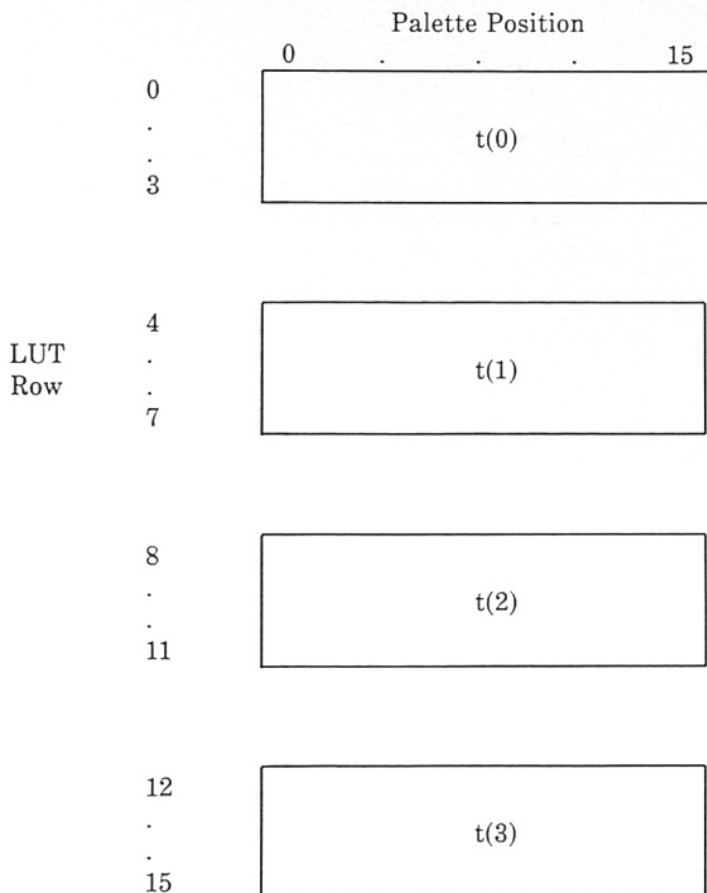
16-COLOR GRAPHICS LUT PROGRAMMING

In these modes, the LUT can be viewed as a two-dimensional array (16 X 16). Each location contains one of the standard 16 colors.



The locations of the LUT are numbered consecutively from left to right and top to bottom. Thus, location 17 corresponds to row 1, palette position 1.

In the 16-color graphics mode, the LUT is divided into four "time states". At any one time, only one quarter of the LUT determines the display on the screen.



The hardware cycles through the LUT every second, so each quarter of the LUT is active for $\frac{1}{4}$ of each second. The cycling mechanism produces blinking. The following examples show the details of how you can produce several different blinking effects by setting different values in the LUT.

In this example, the Write Dot or Write Character functions specify palette position 7 and the LUT is set up as shown. Pixels are displayed as a solid red color. In the first $\frac{1}{4}$ second, the DEB displays the color in the first quarter of the LUT, which in this case is red. In the second, third, and fourth $\frac{1}{4}$ seconds, the DEB displays the color in the second, third, and fourth quarters of the LUT, respectively. In this example, the DEB keeps finding the color value for red, so what you see on the screen is a solid (non-blinking) red color.

		Palette Position			
		LUT			
		Row	0	7	15
t(0)	0		<div><div></div><div>r</div><div>e</div><div>d</div></div>		
	.				
	.				
	3				
t(1)	4		<div><div></div><div>r</div><div>e</div><div>d</div></div>		
	.				
	.				
	7				
t(2)	8		<div><div></div><div>r</div><div>e</div><div>d</div></div>		
	.				
	.				
	11				
t(3)	12		<div><div></div><div>r</div><div>e</div><div>d</div></div>		
	.				
	.				
	15				

Non-Blinking Color

In this example, any item displayed on the screen with palette position 7 blinks between red and blue. For the first $\frac{1}{4}$ seconds, the DEB picks up the color value for red from the first and second quarters of the LUT. For the second two $\frac{1}{4}$ seconds, the DEB obtains the color value of blue from the LUT. The net effect is a slow blink between red and blue.

		Palette Position		
		0	7	15
t(0)	LUT			
	Row			
	0	<div></div>		
	.			
t(1)	.			
	3			
	4	<div></div>		
	.			
t(2)	.			
	7			
	8	<div></div>		
	.			
t(3)	.			
	11			
	12	<div></div>		
	.			
t(4)	.			
	15			

Slow Blink

In this example, any item displayed using palette position 7 blinks rapidly between red, blue, green, and brown.

		Palette Position		
LUT		0	7	15
t(0)	Row			
	0	<div style="border: 1px solid black; padding: 10px; display: flex; align-items: center; justify-content: center;"> r </div>		
	.			
	.			
	3			
t(1)	4	<div style="border: 1px solid black; padding: 10px; display: flex; align-items: center; justify-content: center;"> b </div>		
	.			
	.			
	7			
t(2)	8	<div style="border: 1px solid black; padding: 10px; display: flex; align-items: center; justify-content: center;"> g </div>		
	.			
	.			
	11			
t(3)	12	<div style="border: 1px solid black; padding: 10px; display: flex; align-items: center; justify-content: center;"> b </div>		
	.			
	.			
	15			

4-Color Fast Blink

For dithering colors, the DEB uses a scheme similar to the blinking scheme. Dithering is accomplished by manipulating groups of 4 adjacent pixels. The screen is divided into blocks of 4 pixels. Each of the 4 time states is composed of four rows that determine the dithering effect. The rows of the time state blocks correspond to the 4-pixel blocks on the screen.

The pixels in the pixel blocks are so close together that our eyes cannot perceive them as separate. If each of the pixels in a pixel block is a different color, our eyes perceive the pixel block as one color — a combination of the color of the individual pixels. If the adjacent pixels are the same color, our eyes see just that one color.

Remember the table of “pre-assigned” dithered colors. To combine colors, you check the table for the color number for a particular dither effect.

The following examples show the actual LUT values for some sample cases of blinking and dithering.

		Palette Position		
		0	7	15
t(0)	LUT Row 0	<div>4 (red)</div>		
	1			
	2			
	3			
t(1)	4	<div>4</div>		
	5			
	6			
	7			
t(2)	8	<div>4</div>		
	9			
	10			
	11			
t(3)	12	<div>4</div>		
	13			
	14			
	15			

Palette Position 7 programmed for Non-Blinking Red

		Palette Position			
		LUT	0	7	15
		Row			
t(0)	0		4 (red)		
	1				
	2				
	3				
t(1)	4		4		
	5				
	6				
	7				
t(2)	8		1 (blue)		
	9				
	10				
	11				
t(3)	12		1		
	13				
	14				
	15				

Palette Position 7 programmed to blink slowly between red and blue.

		Palette Position			
		LUT			
		Row	0	7	15
t(0)		0	4 (red)		
		1			
		2			
		3			
t(1)		4	1 (blue)		
		5			
		6			
		7			
t(2)		8	2 (green)		
		9			
		10			
		11			
t(3)		12	6 (brown)		
		13			
		14			
		15			

4-Color Fast Blink

		Palette Position		
		0	7	15
t(0)	LUT Row 0	4 (red)		
	1			
	2			
	3			
t(1)	4	4		
	5			
	6			
	7			
t(2)	8	4		
	9			
	10			
	11			
t(3)	12	4		
	13			
	14			
	15			

Solid Red Dither

		Palette Position			
		LUT			
		Row	0	7	15
t(0)	0		<div>1 (blue)</div>		
	1				
	2				
	3				
t(1)	4		<div>1</div>		
	5				
	6				
	7				
t(2)	8		<div>1</div>		
	9				
	10				
	11				
t(3)	12		<div>1</div>		
	13				
	14				
	15				

2-Color Dither; Red and Blue

		Palette Position			
		LUT			
		Row	0	7	15
t(0)	0		<div><div>4 (red)</div><div>2 (green)</div><div>1 (blue)</div><div>6 (brown)</div></div>		
	1				
	2				
	3				
t(1)	4		<div><div>4</div><div>2</div><div>1</div><div>6</div></div>		
	5				
	6				
	7				
t(2)	8		<div><div>4</div><div>2</div><div>1</div><div>6</div></div>		
	9				
	10				
	11				
t(3)	12		<div><div>4</div><div>2</div><div>1</div><div>6</div></div>		
	13				
	14				
	15				

4-Color Dither Between Red, Green, Blue, and Brown

		Palette Position			
		LUT	0	7	15
		Row			
t(0)		0	<div><div>1 (blue)</div><div>4 (red)</div><div>1</div><div>4</div></div>		
		1			
		2			
		3			
t(1)		4	<div><div>1</div><div>4</div><div>1</div><div>4</div></div>		
		5			
		6			
		7			
t(2)		8	<div><div>2 (green)</div><div>6 (brown)</div><div>2</div><div>6</div></div>		
		9			
		10			
		11			
t(3)		12	<div><div>2</div><div>6</div><div>2</div><div>6</div></div>		
		13			
		14			
		15			

Combination of Blinking and Dithering

The following table of values can be used to program the LUT for normal 16-color graphics.

		Palette Position															
LUT		0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15															
t(0)	Row																
	0	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	1	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	2	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
t(1)	3	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	4	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	5	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	6	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
t(2)	7	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	8	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	9	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	10	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
t(3)	11	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	12	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	13	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	14	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	15	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															

Non-Blinking Standard Colors

		Palette Position															
LUT																	
Row		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t(0)	0	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	1	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	2	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	3	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
t(1)	4	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	5	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	6	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
	7	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,															
t(2)	8	0,1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,															
	9	0,1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,															
	10	0,1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,															
	11	0,1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,															
t(3)	12	0,1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,															
	13	0,1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,															
	14	0,1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,															
	15	0,1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,															

LUT for Blinking Between White and Red in Palette Position 7.

Note: The palette position 7 in the first two time states has been programmed to show white and in the second two time states to show red.

OVERLAY

MODES—LUT

PROGRAMMING

When the LUT is used in the overlay modes it can be viewed as a two-dimensional array with 8 columns and 32 rows. The column values are DEB palette positions. The two values are VDC color values.

In overlay modes, there are 2 separately controlled images: the VDC image and the DEB image. The 2 images are combined on the display screen. Each pixel on the screen has 2 values associated with it: the VDC color and the DEB palette position. The LUT is used to resolve contention between the 2 values associated with each pixel.

The LUT for overlay modes looks like this:

This LUT entry contains the color that will appear on the screen for the particular combination of VDC color and DEB palette position

As in the 16-color graphics modes, the locations in the LUT are numbered consecutively from left to right and top to bottom. For example, location 17 corresponds to Row 2, Palette Position 0.

In the overlay modes, as in the 16-color graphics mode, the LUT is divided into time states that control blinking effects. However, in the overlay modes, the LUT is only divided into two time states. Half of the LUT determines what is being displayed at any time. The top half is used for the first $\frac{1}{2}$ of each second and the bottom half is used for the second $\frac{1}{2}$ of each second.

Using the overlay modes, you create blinking by making the values in the in the top half of the table different from the corresponding values in the bottom half of the table.

		Palette Position							
LUT	Row	0	7
0		t(0)							
.									
.									
15									
16		t(1)							
.									
.									
31									

The following example shows the LUT values for standard Palette 2 of an overlay mode. The LUT is programmed so that the DEB image is displayed only if the VDC color is 0 (black). If the VDC requests any other color, then that color is displayed no matter what the DEB requests. This has the effect of overlaying the VDC image "on top" of the DEB image.

		DEB Palette Position							
VDC Color Values		0	1	2	3	4	5	6	7
t(0)	0	0,	1,	2,	3,	4,	5,	6,	7,
	1	1,	1,	1,	1,	1,	1,	1,	1,
	2	2,	2,	2,	2,	2,	2,	2,	2,
	3	3,	3,	3,	3,	3,	3,	3,	3,
	4	4,	4,	4,	4,	4,	4,	4,	4,
	5	5,	5,	5,	5,	5,	5,	5,	5,
	6	6,	6,	6,	6,	6,	6,	6,	6,
	7	7,	7,	7,	7,	7,	7,	7,	7,
	8	8,	8,	8,	8,	8,	8,	8,	8,
	9	9,	9,	9,	9,	9,	9,	9,	9,
	10	10,	10,	10,	10,	10,	10,	10,	10,
	11	11,	11,	11,	11,	11,	11,	11,	11,
	12	12,	12,	12,	12,	12,	12,	12,	12,
	13	13,	13,	13,	13,	13,	13,	13,	13,
	14	14,	14,	14,	14,	14,	14,	14,	14,
	15	15,	15,	15,	15,	15,	15,	15,	15,

		DEB Palette Position							
VDC									
Color									
Values		0	1	2	3	4	5	6	7
t(1)	0	0,	1,	2,	3,	4,	5,	6,	7,
	1	1,	1,	1,	1,	1,	1,	1,	1,
	2	2,	2,	2,	2,	2,	2,	2,	2,
	3	3,	3,	3,	3,	3,	3,	3,	3,
	4	4,	4,	4,	4,	4,	4,	4,	4,
	5	5,	5,	5,	5,	5,	5,	5,	5,
	6	6,	6,	6,	6,	6,	6,	6,	6,
	7	7,	7,	7,	7,	7,	7,	7,	7,
	8	8,	8,	8,	8,	8,	8,	8,	8,
	9	9,	9,	9,	9,	9,	9,	9,	9,
	10	10,	10,	10,	10,	10,	10,	10,	10,
	11	11,	11,	11,	11,	11,	11,	11,	11,
	12	12,	12,	12,	12,	12,	12,	12,	12,
	13	13,	13,	13,	13,	13,	13,	13,	13,
	14	14,	14,	14,	14,	14,	14,	14,	14,
	15	15,	15,	15,	15,	15,	15,	15,	15,

In this example, the standard Palette 2 is modified so that position 2 is a blinking between blue (color 1) and red (color 4).

		DEB Palette Position								
		VDC								
		Color								
		Values	0	1	2	3	4	5	6	7
t(0)	0		0,	1,	1,	3,	4,	5,	6,	7,
	1		1,	1,	1,	1,	1,	1,	1,	1,
	2		2,	2,	2,	2,	2,	2,	2,	2,
	3		3,	3,	3,	3,	3,	3,	3,	3,
	4		4,	4,	4,	4,	4,	4,	4,	4,
	5		5,	5,	5,	5,	5,	5,	5,	5,
	6		6,	6,	6,	6,	6,	6,	6,	6,
	7		7,	7,	7,	7,	7,	7,	7,	7,
	8		8,	8,	8,	8,	8,	8,	8,	8,
	9		9,	9,	9,	9,	9,	9,	9,	9,
	10		10,	10,	10,	10,	10,	10,	10,	10,
	11		11,	11,	11,	11,	11,	11,	11,	11,
	12		12,	12,	12,	12,	12,	12,	12,	12,
	13		13,	13,	13,	13,	13,	13,	13,	13,
	14		14,	14,	14,	14,	14,	14,	14,	14,
	15		15,	15,	15,	15,	15,	15,	15,	15,

		DEB Palette Position							
	VDC Color Values								
		0	1	2	3	4	5	6	7
t(1)	0	0,	1,	4,	3,	4,	5,	6,	7,
	1	1,	1,	1,	1,	1,	1,	1,	1,
	2	2,	2,	2,	2,	2,	2,	2,	2,
	3	3,	3,	3,	3,	3,	3,	3,	3,
	4	4,	4,	4,	4,	4,	4,	4,	4,
	5	5,	5,	5,	5,	5,	5,	5,	5,
	6	6,	6,	6,	6,	6,	6,	6,	6,
	7	7,	7,	7,	7,	7,	7,	7,	7,
	8	8,	8,	8,	8,	8,	8,	8,	8,
	9	9,	9,	9,	9,	9,	9,	9,	9,
	10	10,	10,	10,	10,	10,	10,	10,	10,
	11	11,	11,	11,	11,	11,	11,	11,	11,
	12	12,	12,	12,	12,	12,	12,	12,	12,
	13	13,	13,	13,	13,	13,	13,	13,	13,
	14	14,	14,	14,	14,	14,	14,	14,	14,
	15	15,	15,	15,	15,	15,	15,	15,	15,

In this example, values in the LUT cause the DEB's output to take precedence over the VDC's output. The VDC's output is only displayed when you specify DEB palette position 0 in a graphics statement.

		DEB Palette Position							
		VDC Color Values	0	1	2	3	4	5	6 7
t(0)	0		0, 1, 2, 3, 4, 5, 6, 7,						
	1		1, 1, 2, 3, 4, 5, 6, 7,						
	2		2, 1, 2, 3, 4, 5, 6, 7,						
	3		3, 1, 2, 3, 4, 5, 6, 7,						
	4		4, 1, 2, 3, 4, 5, 6, 7,						
	5		5, 1, 2, 3, 4, 5, 6, 7,						
	6		6, 1, 2, 3, 4, 5, 6, 7,						
	7		7, 1, 2, 3, 4, 5, 6, 7,						
	8		8, 1, 2, 3, 4, 5, 6, 7,						
	9		9, 1, 2, 3, 4, 5, 6, 7,						
	10		10, 1, 2, 3, 4, 5, 6, 7,						
	11		11, 1, 2, 3, 4, 5, 6, 7,						
	12		12, 1, 2, 3, 4, 5, 6, 7,						
	13		13, 1, 2, 3, 4, 5, 6, 7,						
	14		14, 1, 2, 3, 4, 5, 6, 7,						
	15		15, 1, 2, 3, 4, 5, 6, 7,						

		DEB Palette Position							
		VDC							
		Color							
		Values	0	1	2	3	4	5	6 7
t(1)	1		0, 1, 2, 3, 4, 5, 6, 7,						
	1		0, 1, 2, 3, 4, 5, 6, 7,						
	2		2, 1, 2, 3, 4, 5, 6, 7,						
	3		3, 1, 2, 3, 4, 5, 6, 7,						
	4		4, 1, 2, 3, 4, 5, 6, 7,						
	5		5, 1, 2, 3, 4, 5, 6, 7,						
	6		6, 1, 2, 3, 4, 5, 6, 7,						
	7		7, 1, 2, 3, 4, 5, 6, 7,						
	8		8, 1, 2, 3, 4, 5, 6, 7,						
	9		9, 1, 2, 3, 4, 5, 6, 7,						
	10		10, 1, 2, 3, 4, 5, 6, 7,						
	11		11, 1, 2, 3, 4, 5, 6, 7,						
	12		12, 1, 2, 3, 4, 5, 6, 7,						
	13		13, 1, 2, 3, 4, 5, 6, 7,						
	14		14, 1, 2, 3, 4, 5, 6, 7,						
	15		15, 1, 2, 3, 4, 5, 6, 7,						

The following LUT entirely blocks out VDC output:

		DEB Palette Position							
		VDC							
		Color							
		Values	0	1	2	3	4	5	6 7
t(0)	0		0, 1, 2, 3, 4, 5, 6, 7,						
	1		0, 1, 2, 3, 4, 5, 6, 7,						
	2		0, 1, 2, 3, 4, 5, 6, 7,						
	3		0, 1, 2, 3, 4, 5, 6, 7,						
	4		0, 1, 2, 3, 4, 5, 6, 7,						
	5		0, 1, 2, 3, 4, 5, 6, 7,						
	6		0, 1, 2, 3, 4, 5, 6, 7,						
	7		0, 1, 2, 3, 4, 5, 6, 7,						
	8		0, 1, 2, 3, 4, 5, 6, 7,						
	9		0, 1, 2, 3, 4, 5, 6, 7,						
	10		0, 1, 2, 3, 4, 5, 6, 7,						
	11		0, 1, 2, 3, 4, 5, 6, 7,						
	12		0, 1, 2, 3, 4, 5, 6, 7,						
	13		0, 1, 2, 3, 4, 5, 6, 7,						
	14		0, 1, 2, 3, 4, 5, 6, 7,						
	15		0, 1, 2, 3, 4, 5, 6, 7,						

		DEB Palette Position							
		VDC							
		Color							
		Values	0	1	2	3	4	5	6 7
t(1)	0		0, 1, 2, 3, 4, 5, 6, 7,						
	1		0, 1, 2, 3, 4, 5, 6, 7,						
	2		0, 1, 2, 3, 4, 5, 6, 7,						
	3		0, 1, 2, 3, 4, 5, 6, 7,						
	4		0, 1, 2, 3, 4, 5, 6, 7,						
	5		0, 1, 2, 3, 4, 5, 6, 7,						
	6		0, 1, 2, 3, 4, 5, 6, 7,						
	7		0, 1, 2, 3, 4, 5, 6, 7,						
	8		0, 1, 2, 3, 4, 5, 6, 7,						
	9		0, 1, 2, 3, 4, 5, 6, 7,						
	10		0, 1, 2, 3, 4, 5, 6, 7,						
	11		0, 1, 2, 3, 4, 5, 6, 7,						
	12		0, 1, 2, 3, 4, 5, 6, 7,						
	13		0, 1, 2, 3, 4, 5, 6, 7,						
	14		0, 1, 2, 3, 4, 5, 6, 7,						
	15		0, 1, 2, 3, 4, 5, 6, 7,						

PROGRAMMING THE BIT PLANES Once you have learned to program the LUT directly using the Set Color Palette command, you can make further use of the LUT's capabilities by programming the VDC and DEB video memory directly.

By directly programming the video memory of the VDC and DEB boards, you can increase the graphics display speed. The values you load into the video memory planes determine how the LUT is accessed. This section assumes that you have read and understood how to program the LUT directly.

In the 16-color graphics modes, the device driver combines the 3 bit planes of the DEB with one bit plane from the VDC to create the four bit planes necessary for 16-color graphics.

In the overlay modes, the device driver uses the 3 DEB bit planes for 8-color graphics output and uses the VDC board separately for either text or graphics output.

LUT Addressing

A LUT address is an 8-bit value that points to one of the 256 locations within the LUT. The method of address formation depends on the current video mode.

For transparent and disabled modes, LUT addressing is irrelevant. In the transparent mode, VCD color values bypass the LUT processing and go directly to the monitor output. In the disabled mode, all output from the LUT is forced to the value of zero.

For the 16-color graphics and overlay modes, the LUT address is composed of bits from the DEB video bit planes, the VDC's video output, and DEB timing bits.

Timing Bits

The timing bits are called BLINK1, BLINK2, PAT1, and PAT2. BLINK1 and BLINK2 effect blinking; PAT1 and PAT2 effect patterning (dithering).

All of the timing bits are applicable in the 16-color graphics mode; only BLINK1 is part of the address formation in the overlay mode. Therefore, you have fewer options for blinking and no ability to dither in the overlay mode.

The operation of the timing bits is very fundamental to creation of special effects. The bits always cycle on and off, each at a different rate. BLINK1 cycles on and off each 1/4 second. BLINK2 cycles on and off each 1/8 second.

PAT1 and PAT2 cycle on and off so fast that the eye cannot perceive a blink (PAT1 is the fastest). A dithered color is really 2-4 separate colors that are changing so rapidly that the eye perceives them as one solid color.

PAT1 changes value at the same rate that the monitor's cathode ray moves from one pixel to the next. PAT1's effect on LUT addressing is that it switches the address by 16 LUT entries — in the previous table, between pairs of rows. PAT2 changes value at the same rate that the cathode ray changes scanlines — in the previous table, between one pair of rows and the next pair of rows.

PAT2	PAT1	Portion of LUT
0	0	1st 16 entries of each quarter
0	1	2nd 16 entries of each quarter
1	0	3rd 16 entries of each quarter
1	1	4th 16 entries of each quarter

To output a color to the monitor, the DEB concatenates the DEB timing bits BLINK1, BLINK2, PAT1, PAT2, the BLUE output bit from the VDC, and a bit from corresponding locations on each of the three DEB bit planes.

To output a color to the monitor, the DEB concatenates the following bits: BLINK1, the HIGHLIGHT, GREEN, RED, and BLUE output bits from the VDC, and a bit from corresponding locations on each of the three DEB bit planes.

Short LUT Addresses

The DEB supports a method for you to access only the first sixteen LUT locations. This lets you use normal 16-color graphics without needing to manage all of the 256 LUT locations. You invoke this short addressing mode by a setting bit 2 in AL in the "Set Color Palette" command.

Mouse	11-3
Programming Interface from the GWBASIC Interpreter	11-3
Cursor Descriptions - Text and Graphics Cursors	11-4
Notice to System Programmers	11-5
 List of Mouse Functions	11-7
 Mouse Function Calls	11-8
Mouse Initialization	11-8
Show Cursor	11-8
Hide Cursor	11-9
Get Mouse Position and Button Status	11-9
Set Mouse Cursor Position	11-9
Get Button Press Information	11-10
Get Button Release Information	11-10
Set Minimum and Maximum X Position	11-11
Set Minimum and Maximum Y Position	11-11
Define Graphics Cursor Block	11-12
Define Text Cursor	11-12
Read Mouse Motion Counters	11-13
Define Event Handler	11-14
Light Pen Emulation Mode On	11-15
Light Pen Emulation Mode Off	11-15
Set Mouse Motion/Pixel Ratio	11-15

Mouse

Conditional Hide Cursor	11-16
Get Acceleration	11-16
Set Acceleration	11-17
Set Grid	11-18
Get Mouse Mode	11-18
Set Mouse Mode	11-18
Get Keyboard Emulation Configuration	11-19
Set Keyboard Emulation Configuration	11-20

Mouse

The following provides information to programmers who want to include the AT&T Mouse 6300 in their application programs. This document includes descriptions of the graphics and text cursors, and descriptions of the mouse functions and extended mouse functions.

This protocol is compatible with the protocol of Microsoft Corp. as it is documented in the installation and operation manual published by Microsoft Corp. entitled Microsoft Mouse for the IBM Personal Computer.

The mouse driver software is included with the AT&T Mouse 6300.

The programming interface uses the software interrupt 51. Parameter values are passed and returned via the registers. This interface can be used from assembly language programs or from high level language programs such as C language.

PROGRAMMING It is possible to make a mouse system call from
INTERFACE a GWBASIC program running under the
FROM THE GWBASIC interpreter. The following shows
GWBASIC how to make these calls.
INTERPRETER

Insert an initialization sequence such as:

```
10 DEF SEG=0
20 MSEG=256*PEEK(51*4+3)+PEEK(51*4+2)
30 MOUSE=256*PEEK(51*4+1)+PEEK(51*4)+2
40 DEF SEG=MSEG
```

Be sure that the statements appear before any calls to mouse functions. Then use the CALL statement to make the call.

CALL MOUSE(M1%,M2%,M3%,M4%)

MOUSE is the variable containing the entry offset of the mouse software, and M1%, M2%, M3%, and M4% are the names of the integer variables you have chosen for parameters in this call. They correspond to the values for AX, BX, CX, and DX which are described in this document. As an example:

```
100 'Set minimum and maximum horizontal
    position to
    '(320,100)
200 M1%=7 'function number is 7
300 M3%=0 'minimum coordinate
400 M4%=639 'maximum coordinate
500 CALL MOUSE(M1%,M2%,M3%,M4%)
```

CURSOR DESCRIPTIONS - TEXT AND GRAPHICS CURSORS

The characteristics of the cursor are determined by Functions 9 and 10. AT&T Mouse 6300 has both text and graphics cursors available. The graphics cursor is a shape that moves over the images on the screen. The shape can include an arrow, a sight mark, or others. The software text cursor is a character attribute that moves from character to character on the screen. The hardware text cursor is a moving, sometimes flashing block which also moves from character to character on the screen.

The three cursors are exclusive - only one can be on the screen at one time. You have the option of choosing which cursor will appear on the screen and of switching from one cursor to the other.

Note: When a mouse function refers to the graphics cursor location it is referring to the point on the screen that lies directly under the cursor's target area. The target area is the point in the cursor block that the mouse software uses to determine the cursor coordinates. The upper left corner of the cursor block is used to determine the coordinates.

**NOTICE TO
SYSTEM
PROGRAMMERS**

System programmers who intercept the keyboard hardware interrupt (int 9) or the video I/O software interrupt (int 10H) should exercise utmost care, and note the following:

The mouse driver itself intercepts these two interrupts.

The mouse driver expects to receive all the codes which are relevant for the mouse from the keyboard.

If a program which intercepts the keyboard interrupt is loaded after the driver, the following codes from the keyboard should fall through and be left available for the mouse driver:

77H, 78H, 79H, F7H, F8H, F9H 7AH, 7BH, 7CH, 7DH, FAH, FBH, FCH, FDH, FEH and the next 2 codes which follow an occurrence of FEH (FEH is the prefix character for mouse movement reports).

Codes 74H, 75H, 76H, F4H, F5H, F6H are reserved for future use by mouse software.

All other codes are left by the mouse driver to the next interrupt handler on the chain (presumably the ROM resident keyboard interrupt handler).

With regard to video (int 10H) the program should guarantee that video mode changes (function 0) are intercepted by the mouse driver. The mouse driver guarantees consistency of the screen contents to programs that write on the screen via int 10H (or via DOS).

Some programs write directly on the video memory. In this case the application program should use function 2 (hide cursor) or 16 (conditional hide cursor) before writing, then function 1 (show cursor) after writing. GWBASIC is an example of such a program.

To check for the presence in memory of the AT&T Mouse 6300 mouse driver one should check for a "signature" at the offset 10H from the entry of software interrupt S1. The signature is the string "LOGITECH MOUSE DRIVER".

List of Mouse Functions

The following functions apply to AT&T Mouse 6300. These functions will be described in greater detail in subsequent sections.

Compatible Functions*

Function	Number
Mouse Initialization	0
Show Cursor	1
Hide Cursor	2
Get Mouse Position & Button Status	3
Set Mouse Cursor Position	4
Get Button Press Information	5
Get Button Release Information	6
Set Minimum & Maximum X Position	7
Set Minimum & Maximum Y Position	8
Define Graphics Cursor Block	9
Define Text Cursor	10
Read Mouse Motion Counters	11
Define Event Handler	12
Light Pen Emulation Mode On	13
Light Pen Emulation Mode Off	14
Set Mouse Motion/Pixel Ratio	15
Conditional Hide Cursor	16

AT&T Mouse 6300 Specific Functions

Function	Number
Get Acceleration	29
Set Acceleration	30
Set Grid	31
Get Mouse Mode	32
Set Mouse Mode	33
Get Keyboard Emulation Configuration	34
Set Keyboard Emulation Configuration	35

*These are compatible with the Microsoft Mouse.

Mouse Function Calls

The following descriptions of the mouse functions specify the required input and the expected output of each function. The descriptions also include any special circumstances involved with each function. The input and output parameters are specified as values for registers.

MOUSE INITIALIZA- TION

This function describes whether the mouse hardware and software are installed. The mouse status is 0 if the mouse hardware and software are not installed and -1 if the hardware and software are installed.

Input

AX = mouse status

Output

AX = mouse status
BX = number of buttons

SHOW CURSOR

This function increments the internal cursor flag. If the flag is 0 it displays the cursor on the screen. The cursor tracks the motion of the mouse, changing position as the mouse changes position.

Input

AX = 1

Output

None

HIDE CURSOR

This function removes the cursor from the screen and decrements the internal cursor flag. Although the cursor is hidden it still tracks the motion of the mouse, changing position as the mouse changes position.

Input AX = 2

Output None

GET MOUSE POSITION & BUTTON STATUS

This function reports the status of the buttons. It also reports the position of the cursor horizontally and vertically. The button status is a single integer value with bit 0 representing the left button and bit 1 the right. When the button is down a bit is 1 and when the button is up a bit is 0.

Input AX = 3

Output BX = button status
CX = horizontal cursor position
DX = vertical cursor position

SET MOUSE CURSOR POSITION

This function sets the cursor to the specified horizontal and vertical positions on the screen. The new values must be within the specified ranges of the virtual screen. If you are not using a high resolution screen, the values are rounded to the nearest values permitted by the screen for horizontal and vertical positions.

Input AX = 4
CX = new cursor position (horizontal)
DX = new cursor position (vertical)

Output None

**GET
BUTTON
PRESS
INFORMATION**

This function reports on the current button status. It gives a count of button presses since the last call to Function 5, and it gives the horizontal and vertical position of the cursor the last time the button was pressed.

The BX parameter indicates which button was pressed. If BX is 0, the left button is checked. If BX is 1, the right button is checked. The button status is a single integer value. Bit 0 represents the left button and bit 1 the right. If the button is down a bit is 1 and if it is up a bit is 0.

Input

AX = 5
BX = button

Output

AX = button status
BX = number of button presses
CX = cursor at last press (horizontal)
DX = cursor at last press (vertical)

**GET
BUTTON
RELEASE
INFORMATION**

This function reports on the current button status. It reports a count of the button releases since the last call to Function 6, and it gives the horizontal and vertical position of the cursor since the last time the button was released.

The BX parameter specifies which button is checked. If BX is 0, the left button is checked. If BX is 1, the right button is checked. The button status is a single integer value. Bit 0 represents the left button and bit 1 represents the right button. If a button is down a bit is 1 and if a bit is 0 a button is up.

**SET
MINIMUM &
MAXIMUM
X POSITION**

This function sets the minimum and maximum possible cursor positions on the screen. All cursor movement is restricted to this area once the area is set. The virtual screen determines the maximum and minimum values.

If the cursor is outside the defined area when the call is made, it moves to just inside the area. If the minimum value is greater than the maximum the two values are exchanged.

Input

AX = 7
CX = minimum position
DX = maximum position

Output

None

**SET
MINIMUM &
MAXIMUM
Y POSITION**

This function sets the minimum and maximum possible cursor positions on the screen. All cursor movement is restricted to this area once the area is set. The virtual screen determines the maximum and minimum values.

If the cursor is outside the defined area when the call is made, it moves to just inside the area. If the minimum value is greater than the maximum the two values are switched.

Input

AX = 8
CX = minimum position
DX = maximum position

Output

None

**DEFINE
GRAPHICS
CURSOR
BLOCK**

This function determines what the shape and color of the cursor will be when it is in graphics mode, and it identifies the center of the cursor. The cursor hot spot values must be within the range of -16 to 16. They define one pixel within the cursor.

Input

AX = 9
BX = horizontal cursor hot spot
CX = vertical cursor hot spot
DX = pointer to screen and cursor masks

Output

None

**DEFINE
TEXT
CURSOR**

This function selects the software or hardware cursor. When the hardware cursor is selected this function sets the first and last scan lines which will be shown on the screen. The value of the parameter BX selects the cursor type. If the value is 0, the software text cursor is selected. If the value is 1, the hardware text cursor is selected. When the software text cursor is selected, the values of parameters CX and DX specify the screen and cursor masks. When the hardware cursor is selected, the parameters of CX and DX must contain the line numbers of the first and last scan line in the cursor to be shown on the screen.

Input

AX = 10
BX = select cursor
CX = screen mask value/scan line start
DX = cursor mask value/scan line stop

Output

None

**READ
MOUSE
MOTION
COUNTERS**

Note: In MOUSE.COM there is an accelerator which affects fast mouse movement. Cursor movement is expanded by the accelerator if hand movement is over a certain speed threshold. With slow mouse movement the mouse resolution is 200 Steps per Inch (SPI). With fast mouse movement it is 400 SPI. Information in the following sections changes according to whether the accelerator is in effect or not.

This function returns the horizontal and vertical step count, which is the distance the mouse has moved in 1/200 inch increments, since the last call to this function.

With the Default Accelerator installed the ratio of mouse to cursor movement is variable between 1/200 and 1/400. The step count is always within the range -32768 to 32767. A positive horizontal count specifies a motion to the right while a positive vertical count specifies a motion to the bottom of the screen. The step count is set to 0 after the call is completed.

Input

AX = 11

Output

CX = horizontal count
DX = vertical count

**DEFINE
EVENT
HANDLER**

This function sets the call mask and subroutine address for the mouse software interrupts. The software interrupts stop execution of your program and call the specified subroutine whenever one or more of the conditions defined by the call mask occur. The call mask is a single integer value which defines the conditions which will cause an interrupt.

Each bit in the call mask corresponds to a specific condition:

Mask Bit	Condition
0	change cursor position
1	press left button
2	release left button
3	press right button
4	release right button
5-15	not used

Input

AX = 12
CX = call mask
DX = address offset to subroutine

Output

None

**LIGHT PEN
EMULATION
MODE ON**

This function enables light pen emulation by the mouse. When the mouse emulates the light pen, calls to the pen function will return the cursor position at the last pen down which is controlled by the mouse buttons. The pen is down when both buttons are down. The pen is off the screen when both buttons are up.

Input AX = 13

Output None

**LIGHT PEN
EMULATION
MODE OFF**

This function disables the light pen emulation mode. When light pen simulation is disabled, calls to the pen function return information about the light pen only.

Input AX = 14

Output None

**SET MOUSE
MOTION/PIXEL
RATIO**

This function sets the mouse motion to screen pixel ratio. The horizontal and vertical ratios specify the amount of mouse motion at 8 pixels with the values falling within the range of 1 to 32767. The default values are 8 steps to 8 pixels horizontally and 16 steps to 8 pixels vertically. This is equivalent to 3.2 inches of horizontal mouse movement and 2.0 inches of vertical mouse movement.

Input AX = 15
CX = horizontal step to pixel ratio
DX = vertical step to pixel ratio

Output None

CONDITIONAL HIDE CURSOR This function allows the user to define an area on the screen within which the mouse will turn off. Performing the Show Cursor Function - Function 1 - resets the region and turns the mouse back on. This function is used to guard a portion of the screen which your program is about to update.

Input AX = 16
 CX = left margin
 DX = top margin
 SI = right margin
 DI = bottom margin

Output None

GET ACCELERATION This mode returns the acceleration of the mouse in real mouse mode.

Input AX = 29

Output BX = acceleration type
 ES:DX = acceleration vector address

**SET
ACCELERATION**

This mode establishes the acceleration of the mouse in real mouse mode.

Input

AX = 30
BX = acceleration type
ES:DX = acceleration vector address

Output

None

**Acceleration
type**

BX = 0 default accelerator
BX = 1 use the given accelerator vector
BX = 2 no accelerator

Comments

Acceleration is an array of 128 bytes. Each element contains the value to be substituted as relative mouse movements count when the value used as index for the array is received from the mouse

The Default Acceleration Table is built up in the following way:

Displacements 0,1,2 and 3 are left unchanged.
Displacements 4 through 9 are mapped into 5 through 15.

Displacements 10 through 14 are mapped into 18 through 26.

Displacements 15 through 19 are mapped into 29 through 37.

Displacements 20 through 127 are mapped into 40 through 254.

SET GRID	This function allows the user to set the cursor movements so that they move on the points of a grid on the screen.	
Input	AX = 30 BX = horizontal grid value CX = vertical grid value	
Output	None	
Comments	Grid Value:	0 = 1 = no grid distance of points
	Implementation restriction: Only values which are powers of 2 (2,4,8,16...) are allowed as grid values.	
	The set grid function does not apply in text cursor mode.	
GET MOUSE MODE	This function distinguishes between realmouse mode and keyboard emulation mode. The value returned in BX is 0 when the mode is mousebase and 1 when it is keybase.	
Input	AX = 32	
Output	BX = mode	
SET MOUSE MODE	This function distinguishes between real mouse mode and keyboard emulation mode. A bit is 0 when real mouse mode is set and 1 when keyboard emulation mode is set.	
Input	AX = 33 BX = mode	
Output	None	

**GET
KEYBOARD
EMULATION
CONFIGURA-
TION**

This mode gets the configuration for keyboard emulation mode.

Input

AX = 34

ES:DX = address of parameter structure

Output

ES:DX = parameter structure loaded with configuration values

**Parameter
Structure**

WORD	length of left button press string
DWORD	pointer to left button press string
WORD	length of left button release string
DWORD	pointer to left button release string
WORD	length of middle button press string
DWORD	pointer to middle button press string
WORD	length of middle button release string
DWORD	pointer to middle button release string
WORD	length of right button press string
DWORD	pointer to right button press string
WORD	length of right button release string
DWORD	pointer to right button release string
WORD	length of forward movement string
DWORD	pointer to forward movement string
WORD	length of backward movement string
DWORD	pointer to backward movement string
WORD	length of left movement string
DWORD	pointer to left movement string
WORD	length of right movement string
DWORD	pointer to right movement string
WORD	x scale
WORD	y scale

Comments

Pointers are offset, segment in the 8086 conventions (offset lower word, segment upper word).

Lengths are interpreted as buffer length in input. They will contain actual string lengths in the output.

Length of the button strings cannot exceed 30.

Length of the movement strings cannot exceed 4.

A length of 0 is legal. In this case the corresponding pointer will not be used. The value for the corresponding item will not be returned.

"-1" will be returned in length if the buffer is shorter than the actual string.

**SET
KEYBOARD
EMULATION
CONFIGURA-
TION**

This mode sets the configuration for keyboard emulation mode.

Input

AX = 35

BX = configuration type

ES:DX = address of parameter structure containing configuration values

Output

None

**Configuration
Type**

- 0 = use specified configuration
- 1 = use default configuration;
parameter structure is not used
- 2 = use "reserved" configuration
(for AT&T Mouse 6300 use only);
parameter structure is not used.

**Parameter
Structure**

(See chart for Function 34 - Get Keyboard Emulation Configuration) Pointers are offset, segment in the 8086 conventions (offset lower word, segment upper word).

The length of the button strings cannot exceed 30.

The length of the movement strings cannot exceed 4.

Longer strings will be truncated.

A length of 0 is legal. The corresponding pointer will not be used and no string will be attached to the specific button or movement.

Function 35 will not alter the mouse mode. If a new configuration is issued while the mouse is in keyboard emulation mode, the new configuration will be in effect immediately. If a new configuration is issued while the mouse is in real mouse mode, the new configuration will be in effect at the next setmouse mode which specifies keyboard emulation mode.

The default configuration in keybase mode is keyboard cursor keys for mouse movement, <CR> for the left mouse button, <esc> for the right mouse button, and 5 for both the horizontal and vertical mouse movement scale.



Software Compatibility Considerations	A-4
Six Additional Interrupt Vectors	A-4
Instruction Clock Cycles	A-5
Exception Points to the DIV Instruction	A-6
Numeric Exceptions	A-6
Exception Handlers Prefixes	A-6
Undefined Operations	A-6
Far JMP Instruction at FFFF0H	A-6
The Value Written by PUSH SP	A-7
Shifting More Than 31 Bits	A-7
Duplicate Prefixes	A-7
LOCK Characteristics	A-8
External Interrupt Handlers	A-8
Quotients of 80H or 8000H	A-8
Interrupting NMI Handlers	A-8
 The AT&T PC 6300 PLUS Versus the AT&T PC 6300	 A-9
 MS-DOS 3.1 Versus MS-DOS 2.X	 A-12
New Network Commands	A-12
New General Commands	A-13
New Network Access Calls	A-13
New Network Function Calls	A-14
New Public Function Calls	A-14

General Enhancements	A-15
Installable Device Drivers	A-16
Enhancements to Function Calls and Interrupts	A-18
Miscellaneous Enhancements	A-20
Bug Fixes from MS-DOS 2.11	A-21

This appendix summarizes the differences between the AT&T PC 6300 and the AT&T PC 6300 PLUS, and provides details on how 8086 microprocessor programs are "ported" to the 80286 microprocessor.

Software Compatibility Considerations

In general, the 80286 microprocessor (real mode) will correctly execute ROM-based 8086 microprocessor software. Following is a list of the differences between the 8086 microprocessor and the 80286 microprocessor (real mode).

SIX ADDITIONAL INTERRUPT VECTORS

The 80286 microprocessor adds six interrupts which arise only if the 8086 microprocessor program has a hidden bug. These interrupts occur only for instructions which are undefined for the 8086 microprocessor or if a segment wraparound is attempted. Add an interrupt handler, which will treat these interrupts as invalid operations, to programs for the 80286 microprocessor.

This additional software does not significantly effect existing 8086 microprocessor software, because interrupts normally do not occur and should not have been used since they are in the interrupt group reserved by the manufacturer.

The following table describes the new 80286 microprocessor interrupts:

**Interrupt
Number**

Function

5	A BOUND instruction was executed with a register value outside the two limit values.
6	An undefined opcode was encountered.
7	The EM bit in the MSW has been set and an ESC instruction was executed. This interrupt will also occur on WAIT instructions if TS is set.
8	The interrupt table limit was changed by the LIDT instruction to a value between 20H and 42H. The default limit after reset is 3FFH, enough for all 258 interrupts.
9	A processor extension data transfer exceeded offset OFFFFH in a segment. This interrupt handler must execute FNINIT before any ESC or WAIT instruction is executed.
13	Segment wraparound was attempted by a word operation at offset OFFFFH. A push with SP=1 during PUSH, CALL, or INT will also cause this interrupt.

**INSTRUCTION
CLOCK
CYCLES**

The 80286 microprocessor takes fewer clock cycles than the 8086 microprocessor for most instructions. Areas to watch are delays between I/O operations, and assumed delays in the 8086 microprocessor operating in parallel with an NPX.

**EXCEPTION
POINTS TO
THE DIV
INSTRUCTION**

Interrupts on the 80286 microprocessor will always leave the CS:IP value pointing to the instruction which failed. On the 8086 microprocessor, the CS:IP value saved for a divide exception points to the next instruction.

**NUMERIC
EXCEPTIONS**

The 80286 microprocessor **must** use interrupt vector 16 for the numeric error interrupt. If an 8086 microprocessor uses another vector for the NDP interrupt, both vectors should point to the numeric error interrupt handler.

**EXCEPTION
HANDLERS
PREFIXES**

The CS:IP value saved in the NPX environment will point to any leading prefixes of an ESC instruction. The NDP saved value points to the ESC instruction.

**UNDEFINED
OPERATIONS**

8086 microprocessor instructions like **POP CS** or **MOV cs,opcode** will either cause an exception 6 (undefined opcode) error or perform a protection setup operation like LIDT on the microprocessor. Also, undefined bit encodings for bits 5 through 3 of the second byte of a **POP MEM** or **PUSH MEM** instruction will cause an exception 13 error to occur on the microprocessor.

**FAR JMP
INSTRUCTION
AT FFFF0H**

After reset, CS:IP equals F000:FFF0H on the 80286 microprocessor. This change allows sufficient code space for use in the protected mode without reloading CS.

Place a far JMP instruction at FFFF0H to avoid this difference. Note that the BOOTSTRAP option automatically generates this jump instruction.

**THE VALUE
WRITTEN
BY PUSH SP**

The 80286 microprocessor pushes a different value on the stack for the PUSH SP instruction than the 8086 microprocessor. If this value is important, replace the PUSH SP instructions with the following three instructions:

```
PUSH BP
MOV BP,SP
XCHG BP,[BP]
```

This code emulates the 8086 microprocessor PUSH SP instruction on the 80286 microprocessor.

**SHIFTING
MORE
THAN 31
BITS**

The 80286 microprocessor masks all shift/rotate counts to the lower 5 bits, therefore counts are limited up to 31 bits. With this change, the longest shift/rotate instruction is 39 clocks. Without this change, the longest shift/rotate instruction would be 264 clocks, which delays interrupt response until the instruction completes execution.

**DUPLICATE
PREFIXES**

The 80286 microprocessor instruction length limit is 10 bytes. The only way to violate this limit is by duplicating a prefix two or more times before an instruction. An exception 6 error occurs if the instruction length limit is violated. The 8086 microprocessor has no instruction length limit.

LOCK CHARACTERISTICS

The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. The 80286 microprocessor will assert LOCK during an XCHG instruction with memory (even if the LOCK prefix was not used). LOCK should only be used with the XCHG, MOV, MOVS, INS and OUTS instructions.

The 80286 microprocessor LOCK signal will *not* go active during an instruction pre-fetch.

EXTERNAL INTERRUPT HANDLERS

The priority of the 80286 microprocessor single step interrupt differs from that of the 8086 microprocessor. This change prevents external interrupts from being single-stepped while single stepping through a program. The 80286 microprocessor single step interrupt has higher priority than external interrupts.

The 80286 microprocessor will single step through an interrupt handler invoked by INT instructions or an instruction exception.

QUOTIENTS OF 80H OR 8000H

The 80286 microprocessor can generate the largest negative number as a quotient for IDIV instructions. The 8086 microprocessor will cause exception 0.

INTERRUPTING NMI HANDLERS

After an NMI is recognized, the NMI input and processor extension limit error interrupt is masked until the first IRET instruction is executed.

The NPX error signal does not pass through an interrupt controller, but the NDP INT signal does. Interrupt controller-oriented instructions for the 8086 microprocessor may have to be deleted.

The AT&T PC 6300 PLUS Versus the AT&T PC 6300

For discussion purposes in this appendix, the 8086 microprocessor and the numeric data processor (NDP) are associated with the AT&T PC 6300, while the 80286 microprocessor and the numeric processor extension (NPX) are associated with the AT&T PC 6300 PLUS.

The NPX Versus the NDP

The 80286 microprocessor operating in the real mode executes 8086 microprocessor instructions without major modification. However, because of differences in how the NPX handles numeric exceptions versus the NDP, exception-handling routines **may** need to be changed.

The NPX differs from the NDP as follows:

- The NPX signals exceptions through a dedicated ERROR line to the 80286 microprocessor. However, the NPX error signal does not pass through an interrupt controller as it does with the NDP INT signal. Therefore, all interrupt-controller-oriented instructions in numeric exception handlers for the 8086 microprocessor should be deleted.
- The FENI/FNENI and FDISI/FNDISI NDP instructions do **not** perform useful functions in the NPX. The NPX will ignore these instructions and none of the NPX internal states will be updated.

Even though 8086 microprocessor instructions may be executed on the 80286 microprocessor, it is unlikely that the exception-handling routines containing these instructions will be sent to the NPX.

- Interrupt vector 16 **must** point to the numeric exception-handling routine.
- The ESC instruction address saved in the NPX includes leading prefixes before the ESC opcode. The corresponding address saved in the NDP does not include leading prefixes.
- In protected mode, the format of NPX saved instruction and address pointers is different than for the NDP. The instruction opcode is not saved in protected mode, therefore exception handlers will have to retrieve the opcode from memory.
- Interrupt 7 occurs in the NPX when executing ESC instructions with either task switched (TS) or emulation (EM) of the 80286 microprocessor instruction set (TS=1 or EM=1).??? If TS is set (=1), a WAIT instruction causes interrupt 7. An exception handler should be included in 80286 microprocessor code to handle these situations.
- Interrupt 9 occurs if the second or subsequent words of a floating-point operand falls outside a segment's size.

Interrupt 13 occurs if the starting address of a numeric operand falls outside a segment's size.

An exception handler should be included in 80286 microprocessor code to report these programming errors.

- Except for the processor control instructions, all of the NPX numeric instructions are automatically synchronized by the 80286 microprocessor. The 80286 microprocessor automatically tests the BUSY line of the NPX to ensure that it has completed it's previous instruction before executing the next ESC instruction. No explicit WAIT instructions are required to assure this synchronization.

Explicit WAITs are required for the NDP before each numeric instruction to ensure synchronization.

Although 8086 microprocessor programs having explicit WAIT will execute on the 80286 microprocessor without reassembly, these WAIT instructions are unnecessary.

- Since the NPX does not require WAIT instructions before each numeric instruction, the 80286 microprocessor assembler does not automatically generate these WAIT instructions. The 8086 microprocessor assembler, however, automatically precedes every ESC instruction with a WAIT instruction.

Although numeric routines generated using the 8086 microprocessor assembler generally executes correctly on the 80286 microprocessor, reassembly using the 80286 microprocessor assembler may result in a more compact code.

The processor control instructions for the NPX may be coded using either a WAIT or No-WAIT form of mnemonic. The WAIT forms cause the 80286 microprocessor assembler to precede the ESC instruction with a WAIT instruction; in the identical manner as does the 8086 microprocessor assembler.

MS-DOS 3.1 Versus MS-DOS 2.X

The areas of differences between MS-DOS versions are comprised of both new additions to MS-DOS 3.1 and enhancements to existing MS-DOS 2.X functionality. Let's consider the new features first.

By far the most significant enhancement to MS-DOS 3.1 over version 2.11 is the networking support it provides.

NEW NETWORK COMMANDS

SHARE Command

SHARE is a new command that is a terminate and stay resident module that installs the file-sharing/record-locking mechanism for MS-DOS 3.1 and is useful only when networking is active.

JOIN Command

The JOIN Command joins (splices) the source drive under the destination drive as the correct path. This effectively removes the distinction of the physical drives being separately addressable by their drive letter.

SUBST Command

Creates or substitutes a general string alias for a pathname. This command is used for replacing long path names with a single SUBST command and "virtual" drive letter.

**NEW
GENERAL
COMMANDS****LABEL
Command**

This new command enables the user to add, change, or delete volume names.

**JOIN
Command**

The Join Command joins (splices) the source drive under the destination drive as the correct path. This effectively removes the distinction of the physical drive being separately addressable by their drive letter.

**SUBST
Command**

Creates or substitutes a general string alias for a pathname. This command is used for replacing long path names with a single SUBST command and "virtual" drive letter.

**NEW
NETWORK
ACCESS
CALLS****Lock
Function Call
(5C00H)**

This function with code 00H denies all access (read or write) by any other process to the specified file or region of the file.

**Unlock
Function Call
(5C01H)**

This function with code 02H retrieves the specified entry from the network list of assignments. These assignments typically show the local name of the device, the remote name of the device, and what type of device it is.

**Make Assign
List Entry
Function
(5F003H)**

With code 03H, this function redirects a source device (printer or disk drive) to a network directory (destination device).

**Cancel
Assign List
Entry
Function
(5F004H)**

This function with code 5FH cancels the redirection of a printer or disk drive (source device) to a network directory (destination device).

**NEW
NETWORK
FUNCTION
CALLS**

**Get Machine
Name
Function
(5E00H)**

Function call 5EH code 00H retrieves the netname of the local computer.

**Printer
Setup
Function
(5E02H)**

Function call 5EH code 02H, defines a string of control characters that MS-DOS adds to the beginning of each file sent of the network printer.

**NEW
PUBLIC
FUNCTION
CALLS**

**Get/Set
Allocation
Strategy
Function
(58H)**

This function gets or sets the strategy used by MS-DOS to allocate memory blocks when requested by a process. The strategies available are first fit, best fit, and last fit. The default is first fit.

**Get
Extended
Error
Function
(59H)**

This function call retrieves an extended error code list. The extended codes that are provided describe both the class of the error and an suggested action.

In addition, more detailed hardware codes are provided. If the program does not recognize the extended error code, it should use the more simple MS-DOS 2.x error codes.

**Create
Temporary
File Function
(5AH)**

A program that is running on a network might need to create and write to a temporary, unique named file. This is exactly what this function call accomplishes and is the preferred method to do this.

**Create New
File Function
(5BH)**

This function call is used to create a file in the compatibility mode if one does not already exist. Unlike function 3CH (Create Handle), this function fails if the specified file already exists, rather than truncating it to a length of zero.

**Get Program
Segment
Prefix
Address
Function
(62H)**

This call retrieves the segment address of the 100 hex byte PSP of the currently active process.

**GENERAL
ENHANCEMENTS****The Argv[0]
environment**

One of the most useful features added to MS-DOS 3.1 is the ability to determine where an executable program was loaded from. With many programs utilizing different subdirectories or paths, it is extremely helpful to be able to load overlay files or help files from the original directory. MS-DOS 3.1 now allows this by checking the environment.

Looking past the byte of zeros that terminates the set of environment strings is a set of arguments passed to a program which contains a word count followed by an ASCIIZ string. Programs may use this information to determine where a program was loaded from.

INSTALLABLE DEVICE DRIVERS

Block Device Drivers	The maximum number of block device drivers allowed is now 26. Disk drive letters are assigned alphabetically (A-Z) for each unit in a block device.
Attribute Field	A new attribute has been included in the attribute field of the device header. Bit 11 allows for open/close removable media support. If Bit 11 is set, then removable media call is supported.
Media Check	The return "Error" has been added to Media Check for block devices.
Command Codes	There are three new command codes for the request header. Command code 13 is for device open, code 14 is device close, and code 15 is removable media.
Status Word	A new error code, 0FH, is returned in bit 7 of the status word to indicate an Invalid Disk Change.
New Function Call Parameters	This function call parameters Open or Close, and Removable Media have been added. The Open or Close parameters describes the current file activity on a device. The Removable Media parameter is used by block devices only to determine whether the storage media is removable or not.

- COMMAND.COM** Program invocation has now been changed to allow the specification of a path to a file from the command line (e.g., A>\bin\mp). Also, command such as IF and FOR can use the designation of the path for greater flexibility and control. (e.g., IF [NOT] EXITS [drive:] [path]<file><command>)
- PRINT.COM** New parameters for PRINT allows specification of a path, a device, an internal an buffer size (the default is 512 bytes), the timeslice, the clock ticks, and the que size. Changing the default setting may yield better throughput.
- File Allocation Table** MS-DOS 3.1 will use 16-bit FAT entries if the number of clusters on the disk exceeds 4085. This can significantly reduce "wasted" space on a large hard disk. However, this feature should be used with caution as it can cause significant performance degradation since more allocation information must be buffered and fragmentation will increase. Also, since MS-DOS 3.1 still uses 16-bit sector numbers, the maximum disk size which may be addressed is still 32M (assuming 512 bytes/sector).
- CONFIG.SYS** Two new variables were added to the CONFIG.SYS file for system initialization - LastDrive and FCBs. LastDrive - nn denotes the number of physical drives connected to the system that you may access. The FCBs variable allows the setting of the number of FCBs open at any one time (default value is 4), and the number of FCBs that should remain active given you have exceeded the maximum numbers of FCBs open at any one time set by the first parameter.
- The undocumented SWITCHAR variable does not work at all now.

ENHANCEMENTS TO FUNCTION CALLS AND INTERRUPTS

Open Function Call (3DH)-File Access Modes

This call has had extensive alterations, mainly due to the addition of file sharing/locking attributes for networking. The value of the bits in AL indicate the inherit code, the sharing mode, and the access code.

The inherit code (bit 7) is set as to whether the file is inherited from a child process, or is part of the current process. The sharing mode (bits 4-6) specify what kind of access, if any, other processes have in relation to the opened file. The sharing mode possibilities are compatibility mode, deny both, deny write, deny read, and deny none. The access code (bits 0-3) specify how the opened file is to be used. The access attributes allowed are read access, write access, or both.

IOCTL Function Call (44H)

In addition to the standard IOCTL functionality under MS-DOS 2.x, the IOCTL function call has been expanded to include the following areas:

IOCTL Retry (440BH)

During normal functioning, sharing conflicts may occur. These conflicts are automatically retried for a specified count before being passed on as a critical error. This option allows you to modify that count.

IOCTL Is Redirected Block (4409H)

This option indicates whether a logical device is associated with a network directory.

IOCTL Is Redirected Handle (440AH)

This option indicates whether a handle is for a device across the network or at a location station.

IOCTL Is Changeable (4408H)

This option allows the programmer to determine whether or not this device may support removable media.

**Get Country
Data
Function
(38H)**

Several of the registers contain different data about country dependent information. The format of data pointed to by DS:DX has been increased for future enhancements.

Interrupt 24

A new option (Fail) is specified by a return of 3 in AL from the interrupt 24 handler. This means "return an error on the current system call", or fail the call.

In addition, at the time the interrupt 24H is issued, bits 3-5 in AH (Ignore, Retry, Fail) indicate which responses are allowed.

As previously mentioned, extended errors are available in MS-DOS 3.X, but will be mapped to the existing 1-12 existing error codes if not supported by the application.

MISCELLANEOUS ENHANCEMENTS

Proceed Command

A new command, P (for Proceed), has been added to DEBUG. This command operates like the Trace command, but treats interrupts, subroutine calls, repeat-string-instructions, and loop instructions as a single operation and returns control to the instruction immediately after the called routine. This is extremely useful in tracing across system calls and procedures.

MS-LINKER

A new linker now supports greater than 247 separate segments.

COMMAND.COM

Now looks in a fixed place in order to reload its transient portion whenever a program overlays it. For greater flexibility, you can now designate the COMSPEC variable in the environment to look for the transient portion along a path instead of only the root (e.g., COMSPEC-drive:path/COMMAND.COM).

The Country Variable

The Country variable in CONFIG.SYS initializes the system as to what specific country's date and time format is to be used. It also specifies the currency symbol and the decimal separator. This variable has now been increased to support codes larger than 255.

PRINTF Message Files

The message files are now modularized for improved ease in translating messages into different languages.

**BUG FIXES
FROM MS-
DOS 2.11**

The following "bugs" in MS-DOS 2.11 have been corrected:

- **FORMAT greater than 32K sectors.**
FORMAT will now format between 32K and 64K sectors correctly.
- **NMI (Non Maskable Interrupts).**
All occurrences of loading SP before SS have been eliminated. This allows for use of NMI. IT IS STRONGLY RECOMMENDED THAT NMI NOT BE USED FOR PERIPHERAL DEVICES.
- **Interrupt 24 - Abort.**
When a child process is EXEC'ed, its parent's interrupts 22, 23, and 24 vectors are stored in the 100 hex byte PSP. When the task terminates, (either normally or through interrupt 24 - Abort) the parent of that task is given back control and its vectors are restored from the child's header. In MS-DOS 2.x, Interrupt 24 - Abort was not restoring these vectors. Since COMMAND.COM always resets its own vectors, this problem was not always noticeable when a child of COMMAND.COM'S is terminated.
- **GetFreeSpace (36H) system call.**
This would occasionally get confused under MS-DOS 2.X. This has been fixed.
- **Edlin**
The MOVE and COPY commands would sometimes fail under certain conditions. This has been fixed.

- Several FILE COMPARE bugs have been fixed with a new FC.EXE program.
- FORMAT cannot transfer a system onto a newly formatted disk if the total of system files exceeds 64K bytes. This has been fixed.
- **Redirection**
There was a problem with redirection of input related to Control-C detection. When a program expected to input more characters than were in the input file, the system hung, and you could not control-C out of it. This has been fixed.

Note: This list only constitutes the known bugs fixed at this time.

G

Glossary

The following terms are used in this guide.

.

This abbreviation means "all files on the disk." The command "copy a:*. * b:" means "copy all files from the disk in drive A to the disk in drive B."

Abort

This is a response you can give to MS-DOS when you see a device error message displayed. "Abort" means "stop the program command currently executing." Type "A" for Abort when you see "Abort, Retry, Ignore?" message.

ACE

Abbreviation for asynchronous communications element.

ADRS

Abbreviation for address.

APA

Abbreviation for all points addressable.

Application software

Application software, also called software programs, consist of instructions to the computer which are written in a computer language. Application programs are usually distributed on floppy disks.

Backup disk

A backup disk is a backup of any disk you make with the BACKUP command

BIOS

Abbreviation for basic input/output system.

BPB

Abbreviation for BIOS parameter block.

Bit

A Bit can indicate one of two states usually indicated by "0" or "1".

Byte

A byte is a unit of measurement used by computers. A byte consists of eight "bits" (binary digits). In a binary numbering system, only two marks are used: 0 and 1. Each of these marks is called a binary digit, or a bit. When you type a Dir command, MS-DOS displays filenames and the size of the files. For example, the size of "myfile.txt" may be 6900, which stands for 6900 bytes.

Chapter

A tabbed section in this guide as a chapter in text.

CNTS

Abbreviation for contents.

Command

A command is really a short program that tells MS-DOS how to do a specific task. An example of a command is "Dir", which tells MS-DOS to display a directory listing (the contents of a disk). Some commands are "internal" to MS-DOS; that is, they are not displayed when you display the directory. Other commands, such as Format and Diskcopy, reside on the disk as programs with .EXE extensions. The format program is named FORMAT.EXE, for example.

Control key —

CTRL A control key, usually abbreviated "Ctrl," is a key that allows you to give MS-DOS special commands such as "stop the last command" and "stop the display from scrolling." To use the control key, press the **CTRL** key down at the same time as you press another key. Common control key sequences are **CTRL**C (end a command before it is finished) and **CTRL**S (stop the screen display).

CTRLC

This is a control sequence that stops a command while it is running. **CTRL**C is also used to exit EDLIN insert mode.

CTRLS

This is a control sequence that stops the screen display from scrolling. See also **Control key**.

Copy

This is an MS-DOS command. It copies one or more files from one disk to another, or on the same disk.

Default disk drive

The default disk drive is the drive that MS-DOS searches for any filenames you may type. MS-DOS will look for files in the default drive unless you specify a different default drive. The default drive letter is always displayed with the MS-DOS prompt. For example, if the prompt is A>, "A" is the default drive.

Del

Del is a command that you give to MS-DOS. It is short for "Delete" and tells MS-DOS to delete one or more files. A synonym for Del is "Erase."

Device errors

Device errors are errors that MS-DOS displays while reading or writing to devices on your computer. Devices can be printers, disk drives, and the screen display.

Dir

Dir is a command that you give to MS-DOS. It is short for "Directory." When you type "dir", MS-DOS will display the contents of the disk on the default drive. The command "dir b:" displays the contents of the disk in drive B.

Directory

A directory is a table of contents for a floppy disk. The directory contains the names of your files, and also has information on the size of the files and the dates they were created or last modified.

Disk

See Floppy disk and Fixed disk.

Diskcopy

Diskcopy is an MS-DOS command used to copy diskettes.

Disk drive

A disk drive is a piece of hardware that is attached to your computer. A disk drive can be either a floppy or a fixed drive. You insert floppy disks into floppy disk drives; fixed disk drives are built into the computer. A fixed disk can hold more information than a floppy disk.

Disk drives are commonly referred to as the "A" drive and the "B" drive. Fixed disks are usually the "C" drive. Your computer manual should tell you which drive is labeled drive A and which drive is drive B. If you have only one disk drive, that is drive A.

Disk operating system

A disk operating system is a group of programs that act as a translator between you and your computer. A disk operating system is usually distributed on floppy disks, although you can copy the programs to a fixed disk if you have one.

Display

A display looks a lot like a television screen. It attaches to your computer (or may come already attached) so that you can communicate with MS-DOS.

Double word

A 32-bit or 2-word unit of information, typically indicating a complete address which includes the segment and offset.

Drive name

A drive name consists of a drive letter and a colon. A drive name tells MS-DOS what drive to look on for the file. For example, the filename "a:myfile.txt" contains a drive name (a:) that tells MS-DOS to look on drive A for the file named "myfile.txt."

ECC

Abbreviation for error correcting code.

Editor

An editor is a program that allows you to type text and data on the computer. All editors allow you to move, add and delete characters and lines, and save files. The MS-DOS editor is called EDLIN. This is a line-oriented editor, which means that you can only process text one line at a time. Other types of editors are called screen-oriented editors. These allow you to operate on large portions of text (even the entire file) at a time. Generally speaking, word processing programs are powerful, screen-oriented editors.

EDLIN

EDLIN is a line-oriented editor that comes with MS-DOS. See also Editor.

Enter key — **ENTER** or **RETURN**

The Enter key is marked "Return" on some computers. This key is usually pressed after entering data or text. It is also pressed after you have typed a command to MS-DOS.

EOF

Abbreviation for end-of-file.

EOI

Abbreviation for end of interrupt.

Erase

Erase is a synonym for the MS-DOS Del command. See also Del.

Error messages

Error messages are displayed on the screen if MS-DOS detects that something went wrong when it tried to process a command or program. Refer to Appendix A, "Messages," for the appropriate response to each error message.

Extension

A file extension is from 1-3 characters long and starts with a period. Extensions are often used to identify files: most application programs supply their own extensions to files they create. All BASIC files use an extension of ".BAS." See also Filename.

FAT

Abbreviation for file allocation table.

FCB

Abbreviation for file control block.

File

A file is a collection of related information. A file on a disk can be compared to a file folder in a desk drawer. For example, a file folder might contain the names and addresses of your friends. You might name this file "Friends." A file on a disk could also contain the names and addresses of friends. This file could also be named "friends." Programs are also files.

Filename

There are certain rules for naming files on a disk. Names can be from 1-8 characters long, and they can have a file extension. An extension is from 1-3 characters long and starts with a period (.). Together, the file name and extension are referred to as filename. An example of a filename is "myfile.txt". Certain filenames are reserved by MS-DOS and should not be used when naming your files. These filenames are:

aux
con
lst
prn
nul

Fixed disk

A fixed disk is a disk that is built into the computer. A fixed disk can store much more information than a floppy disk, and the computer can retrieve information from it faster. See also Floppy disk.

Floppy disk

A floppy disk is a plastic square that consists of a disk (inside) sealed into a protective cover (outside). There is often a write-enable notch on the right side of the disk. A floppy disk is used for storing programs and files. Disks can be either "single-sided" or "double-sided." Single-sided disks store files on only one side of the disk; double-sided disks store information on both sides of the disk.

When you insert a floppy disk in the computer, the disk drive reads the magnetic surface of the disk. This is how information is passed from a disk into the computer's memory. See also Write-protect notch, Memory.

Format

Format is an MS-DOS command that formats blank disks. You must format every disk before it can be used with MS-DOS. Formatting a disk changes it to a format that MS-DOS can use. It also analyzes the disk for defective spots.

GWBASIC

GWBASIC stands for GW Beginner's All-purpose Symbolic Instruction Code. GWBASIC is a general-purpose computer language and is the first computer language that many people learn.

Ignore

This is a response to a device error message. It tells the computer to ignore the error and continue processing. Note that this response can damage data on the disk. Type "I" for Ignore when you see the "Abort, Retry, Ignore?" message.

H

Symbol placed after a number to refer to its value in Hexadecimal. "H" is used in text when it is not immediately obvious that the number is in Hexadecimal.

ID

Abbreviation for identification.

Input

Input is information given to the computer. This information can come from the keyboard (when you type commands), programs, and even other computers. See also Output.

INT

Abbreviation for interrupt. Used extensively in Chapter 7 to distinguish interrupts from functions.

I/O

Input/Output—Common reference used in text when discussing input and/or output devices.

K

Symbol placed after a number to refer to the value “thousand” when discussing size in bytes, but actual value is 1024. For example, if memory is listed at 64K, that is referred to as “64 thousand” but is actually 65,536 bytes.

LSB,MSB

Least significant byte, most significant byte—The terms LSB and MSB are used to indicate which byte of a 2-byte word is being discussed. The LSB is bits 0 through 7 of the word and the MSB is bits 8 through 15.

M

Symbol placed after a number to refer to the value “million” when discussion size in bytes, actual value is 1,048,576 or 1024 K bytes.

Memory

Memory is synonymous with “computer storage.” Most programs come on a floppy or fixed disk. They are then transferred into the computer's memory (internal storage) when you run the program. When the program is finished, it is transferred back to the disk. Since a computer's memory is limited, this is an efficient way to run many different programs.

Memory is measured in kilobytes (see also byte). Computers that run MS-DOS commonly have 128K of memory or more.

MS-DOS master disk

MS-DOS is distributed on one or more floppy disks (called “master disks”) along with the user's manuals. You should always make a working copy of the master disk or disks before you start using MS-DOS on a routine basis. See also Backup disk.

N/A

Not applicable—Common reference used to indicate that the item does not apply in this particular situation.

Name

See filename.

Nibble

A 4-bit unit of information. A Nibble can indicate 16 different states usually indicated by 0 through FH.

NPX

Abbreviation for numeric processor extension.

Operating system

An operating system is a group of programs that provides the interface between you (the user) and the hardware (your computer). An operating system translates your commands to the computer so that you can perform tasks such as creating files, running programs, and printing documents.

Output

Output is information that is transferred from the computer to any "output device." Output devices are printers, disk drives, and the screen (display). An error message, for example, is output.

Pathname

[d:]path[name].[ext]—the entire file entry which contains the drive number, path, name, and extension; for example, B:/sys/mydir myprog.exe

Print

This is an MS-DOS command that is used to Spool print files to your computer's printer.

Printer

A printer is a device that is attached to your computer. It allows you to print files so that you have a "hard copy" (paper copy or printout) of the information to store.

Program

A program is a complete set of instructions, written in computer language, that tells the computer exactly how to handle a problem. Some commands in MS-DOS, like Diskcopy, are actually small programs. Programs are stored as files on a disk, and usually have special filename extensions to identify them as programs. A common program file extension is ".COM" (for "command") or ".EXE" (for "executable").

Prompt

The MS-DOS prompt consists of the default drive letter (usually A, B, or C) and a greater-than sign. An example of the MS-DOS prompt is B>.

PSP

Abbreviation for program segment prefix.

RAM

Abbreviation for random access memory.

Rename

This is an MS-DOS command. It is used to rename files. The abbreviation "Ren" can be used in place of the full command name.

Retry

This is a response you can give to MS-DOS in response to "Abort, Retry, Ignore?" following a device error message. Retry means "retry the last command." Usually, you will choose Retry only after you have performed a corrective action (such as closing the disk drive door or inserting the proper disk in the drive). Type "R" for Retry when you see the "Abort, Retry, Ignore" message.

Return key — **RETURN** or **ENTER**

The Return key is marked "Enter" on some computers. This key is usually pressed after entering data or text. It is also pressed after you have typed a command to MS-DOS.

ROM

Abbreviation for read only memory.

Software

Software is the internal programs or routines written by programmers to simplify programming and computer operations. These routines allow the programmer to use his own language (English) or mathematics (Algebra) to communicate with the computer. Some examples of software are: operating systems, word processing programs, and spreadsheet programs. Software is also called programs.

Type

This is an MS-DOS command used to display files on the screen.

UART

Abbreviation for Universal Asynchronous Receiver/Transmitter.

Word

A 16-bit or 2-byte unit of information, typically indicating an address such as a segment or offset.

Word processing

Word processing generally involves using an editor and a text processing program to manipulate text and data in files. (Some word processing programs include both functions.) Word processing allows you to type text and then reformat it—for example, into columns or double-spaced lines. See also Editor.

Write-enable notch

Some floppy disks are protected; that is, you can examine information on the disk but you cannot change it. These disks are called "write-protected" disks. They usually have a small tab (a "write-protect tab") covering a notch on the right side of the disk. You can copy information onto the disk by removing the tab first. If the disk does not have a write-enable notch, you cannot change the information on the disk.

Write-protect tab

The small tab that covers the write-enable notch on the disk. See also Write-enable notch.

A

A (ASSEMBLE), 3-12
Abbreviations Used in This Guide, 1-10
Acceleration type, 11-17
ACE, 1-10
Addressing Modes, 4-13
 80286 Microprocessor Instruction Set, 4-17
 Based Addressing, 4-16
 Based Indexed Addressing, 4-17
 Based Indexed with Displacement Addressing, 4-17
 Direct Addressing, 4-15
 Immediate Operand Addressing, 4-14
 Indexed Addressing, 4-16
 Memory Addressing, 4-14
 Register Indirect Addressing, 4-16
 Register Operand Addressing, 4-14
 Register and Immediate Addressing, 4-14
Addressing Scheme, 4-1
 Addressing Modes, 4-13
 Flags, 4-10
 Memory Addressing, 4-3
 Overview, 4-2
 Registers, 4-8
ADRS, 1-10
AF, 4-10
Aligned and Non-Aligned Words, 4-7
Allocate Memory, 7-188
Allocation Block, 6-13
Alphabetical Table of Functions, 7-12
APA, 1-10
Argv[0] environment, A-15
ASCII Strings, 5-8
Asynchronous Communications Element (ACE), 9-36
 Break Control Feature, 9-43
 Example, 9-43
 Functions, 9-37
 Interrupt Priority, 9-44

Asynchronous Communications Element (ACE) (Contd)
 Registers, 9-38
 Sequencing and Timing, 9-42
 Setting the Baud Rate, 9-42
AT&T PC 6300 PLUS Versus the AT&T PC 6300, A-9
Attribute, 9-16
Attribute Field, A-16
Automatic Response File Entry, 2-18
Auxiliary Input, 7-41
Auxiliary Output, 7-42

B

Background, 9-8
 Clock Device, 9-12
 Fixed Addressing, 9-9
 I/O Instructions, 9-9
 I/O Ports, 9-9
 Interrupt Devices, 9-12
 Interrupts, 9-10
 Programmable Devices, 9-8
 Variable Addressing, 9-10
Based Addressing, 4-16
Based Indexed Addressing, 4-17
Based Indexed with Displacement Addressing, 4-17
BF, 3-57
BIOS, 1-10
BIOS Routines, 8-1
 Bootstrap Routine, 8-45
 Bypassing The BIOS Routines, 8-47
 Communication Routines, 8-27
 Disk Routines, 8-22
 Equipment Check Routine, 8-19
 Keyboard Routines, 8-32
 Overview, 8-4
 Print Screen Routine, 8-7
 Printer Routines, 8-43

BIOS Routines (Contd)

- Protected Mode Memory Size Routine, 8-31
- ROM BIOS Listing, 8-48
- Real Mode Memory Size Routine, 8-21
- Routine Interrupt Table, 8-6
- Routines, 8-5
- Time-Of-Day Routine, 8-46
- Video Routines, 8-8
- Bit, 1-8
- Blinking Color Effects for DEB Palettes 0-3, 10-26
- Block Device Drivers, A-16
- Block Devices, 9-6
- BP, 3-57
- BPB, 1-10
- BR, 3-57
- Break Control Feature, 9-43
- Buffer Contents, 7-55
- Buffered Keyboard Input, 7-54
- Bug Fixes from MS-DOS 2.11, A-21
- Bus Interface Devices, 9-50
- Bypassing The BIOS Routines, 8-47
- Byte, 1-8

C

- C (COMPARE), 3-16
- Calling From Macro Assembler, 7-15
- Calling from a High-Level Language, 7-15
- Cancel Assign List Entry, 7-242
- Cancel Assign List Entry Function (5F004H), A-14
- CF, 4-11
- Change Directory Entry, 7-209

- Change the Current Directory, 7-146
- Changing Diskettes, 2-6
- Character Devices, 9-6
- Character Handling, 8-18
- Check Keyboard Status, 7-57
- Check Keystroke, 8-32
- Class, 2-8
- Clock Device, 9-12
- Close File, 7-67
- Close Handle, 7-152
- Cluster Number, 5-23
- Clusters, 5-16
- CNTS, 1-10
- Color Graphics Mode, 8-17
- Color Text Modes, 8-16
- Command Code, 9-21
- Command Codes, A-16
- Command Line Entry, 2-17
 - Automatic Response File Entry, 2-18
 - Syntax, 2-17
 - lib-list, 2-17
 - listfile, 2-17
 - obj-list, 2-17
 - runfile, 2-17
 - /switch, 2-17
- Command Protocol, 9-55
- Command and Statement Text Syntax, 1-4
- COMMAND.COM, A-17, A-20
 - 01-4,
- Communications Manager Interface, 9-45
 - Bus Interface Devices, 9-50
 - Command Protocol, 9-55
 - Comments, 9-50
 - Functions, 9-46
 - Intel 8251 UART Pin Assignments, 9-54
 - Interrupts, 9-50
 - National 8250 UART Pin Assignments, 9-53

Compatibility, A-1
 MS-DOS 3.1 Versus MS-DOS
 2.X, A-12
 Software Compatibility
 Considerations, A-4
 AT&T PC 6300 PLUS Versus the
 AT&T PC 6300, A-9
Conditional Hide Cursor, 11-16
CONFIG.SYS, A-17
Configuration Type, 11-21
Cons for .COM, 6-4
Cons for .EXE, 6-3
Control Flags, 4-12
Control-C Check, 7-130
Control-C Exit Address, 6-13
Controller Internal Diagnostic
Function, 9-133
Conventions Used in This Guide, 1-4
 Command and Statement Text
 Syntax, 1-4
 Displaying of Prompts and
 Messages, 1-6
 Examples, 1-6
 Type Styles, 1-5
Conversions, 6-7
Country Variable, A-20
Create Directory, 7-142
Create File, 7-82
Create Handle, 7-148
Create New File, 7-224
Create New File Function (5BH), A-15
Create New PSP, 7-106
Create Temporary File, 7-221
Create Temporary File Function
(5AH), A-15
CS Value, 6-11
Current Block Number, 5-11

Current Record Number, 5-14
Cursor Descriptions—Text and
Graphics Cursors, 11-4
Customized LUT Input, 10-23

D

D (DISPLAY), 3-18
Data Length, 9-94
Data Register Commands, 9-103
DEB Capabilities, 10-3
 16-Color Graphics, 10-5
 DEB Driver, 10-4
 Look-Up Table (LUT), 10-6
 Overlay Modes, 10-6
DEB Driver, 10-4
DEB Settings, 10-15
DEBUG Commands, 3-2
DEBUG Error Messages, 3-57
 BF, 3-57
 BP, 3-57
 BR, 3-57
 DF, 3-57
DEBUG Program, 3-1
 A (ASSEMBLE), 3-12
 C (COMPARE), 3-16
 D (DISPLAY), 3-18
 DEBUG Commands, 3-2
 DEBUG Error Messages, 3-57
 E (ENTER), 3-20
 F (FILL), 3-24
 G (GO), 3-26
 H (HEX ARITHMETIC), 3-29
 How to Start DEBUG, 3-7
 I (INPUT), 3-31
 L (LOAD), 3-32
 M (MOVE), 3-35
 N (NAME), 3-37
 O (OUTPUT), 3-40

DEBUG Program (Contd)

- Overview, 3-3
- P (PROCEED), 3-41
- Q (QUIT), 3-42
- R (REGISTER), 3-44
- S (SEARCH), 3-48
- T (TRACE), 3-50
- Terms Used in DEBUG, 3-4
- U (UNASSEMBLE), 3-52
- W (WRITE), 3-54
- Default Palettes, 10-24
- Define Event Handler, 11-14
- Define Graphics Cursor Block, 11-12
- Define Text Cursor, 11-12
- Delete File, 7-74
- Delete a Directory Entry, 7-158
- Destination String, 7-240
- Device Drivers, 9-1
 - Asynchronous Communications Element (ACE), 9-36
 - Background, 9-8
 - Communications Manager Interface, 9-45
 - DMA Controller, 9-82
 - Device Headers, 9-15
 - Display Controller, 9-66
 - Floppy Diskette Controller (FDC), 9-92
 - Hard Disk Unit Controller, 9-118
 - Hardware Controller Documentation, 9-35
 - Keyboard Interface, 9-139
 - Mouse (Optional), 9-144
 - Overview, 9-5
 - Parallel Printer Interface, 9-147
 - Programmable Interrupt Controller (PIC), 9-151
 - Programmable Interval Timer, 9-161
 - Real-Time Clock And Calendar, 9-166
 - Request Header, 9-20

Device Drivers (Contd)

- Speaker, 9-172
- Switching Between Real and Protected Modes, 9-175
- Device Headers, 9-15
 - Attribute, 9-16
 - Device Name or Size, 9-19
 - Format, 9-15
 - Pointer to Device Interrupt Entry Point, 9-19
 - Pointer to Device Strategy Entry Point, 9-19
 - Pointer to Next Device, 9-15
- Device Name or Size, 9-19
- DF, 3-57, 4-12
- Direct Addressing, 4-15
- Direct Console I/O, 7-46
- Direct Console Input, 7-49
- Disabled Mode, 10-11
- Disk Boot Sector, 5-24
 - Bytes per Sector, 5-25
 - Format, 5-24
 - JUMP to Boot Code, 5-25
 - Media Descriptor, 5-26
 - Number of FAT Sectors, 5-26
 - Number of FATs, 5-25
 - Number of Heads, 5-26
 - Number of Hidden Sectors, 5-26
 - Number of Root Directory Entries, 5-25
 - Number of Sectors in Logical Image, 5-25
 - OEM Name and Version, 5-25
 - Reserved Sectors, 5-25
 - Sectors per Allocation Unit, 5-25
 - Sectors per Track, 5-26
- Disk Directory, 5-17
 - Cluster Number, 5-23
 - File Attribute, 5-18
 - File Size, 5-23
 - Filename Extension, 5-18

Disk Directory (Contd)
 Format, 5-17
 Last Change Date, 5-21
 Last Change Time, 5-20
 Name or Status, 5-17
 Reserved, 5-20
Disk Formats, 5-33
 Formats, 5-33
Disk Layout, 5-16
 Clusters, 5-16
Display Character, 7-40
Display Controller, 9-66
 Example, 9-81
 Functions, 9-68
 Graphics Mode, 9-77
 Modes, 9-74
 Registers, 9-68
 Sequencing and Timing, 9-80
 Text Mode, 9-74
Display Enhancement Board, 10-1
 DEB Capabilities, 10-3
 How To Program The DEB, 10-10
 Interrupt 10H Functions, 10-13
 Programming Tips, 10-8
 Programming the LUT, 10-30
Display String, 7-53
Displaying Graphics Images, 10-12
Displaying of Prompts and
 Messages, 1-6
Dither Combinations for DEB Palettes
 0-3, 10-27
DMA Controller, 9-82
 Functions, 9-84
 Registers, 9-86
 Sequencing and Timing, 9-91
Document Conventions, 1-4
Double Word, 1-9
Drive Diagnostic Function, 9-132
Drive Number, 5-11

/DSALLOCATE, 2-13
Duplicate File Handle, 7-182
Duplicate Prefixes, A-7

E

E (ENTER), 3-20
ECC, 1-10
End Process, 7-201
End-of-Track, 9-94
Enhancements to Function Calls and
 Interrupts, A-18
EOF, 1-10
EOI, 1-10
Error Codes, 7-22
Error Exit Address, 6-13
Error Messages, 6-8
Exception Handlers Prefixes, A-6
Exception Points to the DIV
 Instruction, A-6
EXE Files, 3-8
EXE2BIN Program, 6-6
 Comments, 6-6
 Conversions, 6-7
 Destination, 6-6
 Error Messages, 6-8
 Source, 6-6
 Syntax, 6-6
Ext, 2-5
Extended FCB, 7-70
Extended FCB Format, 5-14
Extended Indicator, 5-15
Extension, 1-9
External Interrupt Handlers, A-8

F

- F (FILL), 3-24
- Far JMP Instruction at FFFF0H, A-6
- FAT, 1-10
- FAT Entries, 5-28
- FCB, 1-11
- File Allocation Table, A-17
- File Allocation Table (FAT), 5-27
 - Example, 5-32
 - FAT Entries, 5-28
 - Format, 5-27
 - How to Use the FAT, 5-30
 - Media Descriptor, 5-27
 - NNN, 5-29
- File Attribute, 5-15, 5-18
- File Control Blocks, 5-10
 - Current Block Number, 5-11
 - Current Record Number, 5-14
 - Drive Number, 5-11
 - Extended FCB Format, 5-14
 - Extended Indicator, 5-15
 - File Attribute, 5-15
 - File Size, 5-12
 - Filename Extension, 5-11
 - Last Change Date, 5-12
 - Logical Record Size, 5-12
 - Name, 5-11
 - Relative Record Number, 5-14
 - Reserved, 5-14, 5-15
 - Standard FCB, 5-15
 - Standard FCB Format, 5-10
 - # of Bytes, 6-10
 - # of Overlays, 6-11
 - # of Relocation Entries, 6-10
- File Header Format, 6-9
 - CS Value, 6-11
 - Format, 6-9
 - IP Value, 6-11
 - Maximum#, 6-10
- File Header Format (Contd)
 - Minimum#, 6-10
 - Relative Offset, 6-11
 - SP Value, 6-11
 - Size of File, 6-10
 - Size of Header, 6-10
 - Stack Value, 6-10
 - Sum of Words, 6-11
- File Size, 5-12, 5-23
- Filename, 1-9, 2-4
- Filename Extension, 5-11, 5-18
- Find First File, 7-204
- Find Next File, 7-206
- Fixed Addressing, 9-9
- Fixed Disk, 8-24
- Flags, 4-10
 - AF, 4-10
 - CF, 4-11
 - Control Flags, 4-12
 - DF, 4-12
 - IF, 4-12
 - IOPL, 4-12
 - NT, 4-12
 - OF, 4-11
 - PF, 4-11
 - SF, 4-11
 - Status Flags, 4-10
 - TF, 4-12
 - ZF, 4-11
- Floppy Diskette Controller (FDC), 9-92
 - Comments, 9-114
 - Data Length, 9-94
 - Data Register Commands, 9-103
 - End-of-Track, 9-94
 - Gap Length, 9-94
 - Interrupts, 9-117
 - Registers, 9-93
 - Sequencing and Timing, 9-114
- Flush Buffer, Read Keyboard, 7-59

Force Duplicate of a Handle, 7-184
Format Bad Track Function, 9-127
Format Drive Function, 9-126
Format Track Function, 9-127
Free Allocated Memory, 7-190
Function Dispatch Call, 6-13
Functions, 7-35
 Registers, 7-35

G

G (GO), 3-26
Gap Length, 9-94
General Documentation—AT&T
Personal Computer 6300 PLUS, 1-18
General Enhancements, A-15
General Macros, 7-245
General Registers, 4-8
Get Acceleration, 11-16
Get Assign List Entry, 7-236
Get Button Press Information, 11-10
Get Button Release Information, 11-10
Get Communications Port Status, 8-30
Get Country Data Function (38H), A-19
Get Current Directory Path, 7-186
Get Current Disk, 7-86
Get Current Shift Status, 8-33
Get Date, 7-117
Get Default Drive Data, 7-89
Get Disk Free Space, 7-134
Get Disk Transfer Address, 7-127
Get Drive Data, 7-91
Get Extended Error, 7-217
Get Extended Error Function
(59H), A-15
Get File Size, 7-99
Get Interrupt Vector, 7-132

Get Keyboard Emulation
Configuration, 11-19
Get MS-DOS Version Number, 7-128
Get Machine Name, 7-232
Get Machine Name Function
(5E00H), A-14
Get Mouse Mode, 11-18
Get Mouse Position and Button
Status, 11-9
Get PSP, 7-244
Get Printer Port Status, 8-44
Get Program Segment Prefix Address
Function (62H), A-15
Get Return Code of Child
Process, 7-203
Get Time, 7-121
Get Verify State, 7-208
Get/Set Allocation Strategy, 7-214
Get/Set Allocation Strategy Function
(58H), A-14
Get/Set Attributes, 7-163
Get/Set Country Data, 7-136
Get/Set Date/Time of File, 7-211
Graphics Mode, 9-77
Group, 2-7
GW BASIC, 7-15

H

H, 1-11
H (HEX ARITHMETIC), 3-29
Handles, 5-9
 Pre-Defined Handles, 5-9
Hard Disk Unit Controller, 9-118
 Controller Internal Diagnostic
 Function, 9-133
 Drive Diagnostic Function, 9-132
 Example, 9-138
 Format, 9-121

Hard Disk Unit Controller (Contd)
 Format Bad Track Function, 9-127
 Format Drive Function, 9-126
 Format Track Function, 9-127
 Functions, 9-121
 Initialize Drive Characteristics
 Function, 9-129
 Parameters, 9-122
 RAM Diagnostic Function, 9-132
 Read ECC Burst Error Length
 Function, 9-130
 Read Function, 9-128
 Read Long Function, 9-133
 Read Sector Buffer Function, 9-131
 Read Verify Function, 9-126
 Recalibrate Function, 9-124
 Registers, 9-118
 Request Sense Function, 9-124
 Seek Function, 9-129
 Sequencing and Timing, 9-134
 Test Drive Ready Function, 9-123
 Write Function, 9-128
 Write Long Function, 9-134
 Write Sector Buffer Function, 9-131
 Hardware Controller
 Documentation, 9-35
 Hardware/Software
 Compatibility, 10-8
 HEX Files, 3-8
 Hide Cursor, 11-9
 /HIGH, 2-14
 High Resolution Monochrome
 Graphics, 8-17
 How To Program The DEB, 10-10
 16-Color Graphics Modes, 10-11
 Disabled Mode, 10-11
 Displaying Graphics Images, 10-12
 Mode Setting, 10-10
 Overlay Modes, 10-11
 Setting Colors and Effects, 10-12
 Transparent Mode, 10-11

How to Create a Device Driver, 9-7
 How to Install a Device Driver, 9-7
 How to Start DEBUG, 3-7
 Comments, 3-7
 EXE Files, 3-8
 Examples, 3-8
 Filename, 3-7
 HEX Files, 3-8
 Syntax, 3-7
 arglist, 3-7
 How to Use the FAT, 5-30

I

I (INPUT), 3-31
 ID, 1-11
 IF, 4-12
 Immediate Operand Addressing, 4-14
 Indexed Addressing, 4-16
 Initial Conditions, 6-17
 Initialize Communication Port, 8-27
 Initialize Drive Characteristics
 Function, 9-129
 Initialize Printer Port, 8-44
 Installable Device Drivers, A-16
 Instruction Clock Cycles, A-5
 INT, 1-11
 INT 10H—Video Routines, 8-8
 Character Handling, 8-18
 Color Graphics Mode, 8-17
 Color Text Modes, 8-16
 Comments, 8-16
 High Resolution Monochrome
 Graphics, 8-17
 Monochrome Graphics, 8-18
 Monochrome Text Mode, 8-16
 Read Attribute or Character, 8-12
 Read Current Video State, 8-15
 Read Cursor Position, 8-10

-
- INT 10H—Video Routines (Contd)
 - Read Light Pen Position, 8-10
 - Read Pixel, 8-15
 - Scroll Active Page Down, 8-12
 - Scroll Active Page Up, 8-11
 - Select Active Page Number, 8-11
 - Set Color Palette, 8-14
 - Set Cursor Position, 8-10
 - Set Cursor Type, 8-9
 - Set Mode and Clear Screen, 8-9
 - Write Attribute or Character, 8-13
 - Write Character, 8-13
 - Write Pixel, 8-14
 - Write Teletype, 8-15
 - INT 11H—Equipment Check Routine, 8-19
 - INT 12H—Real Mode Memory Size Routine, 8-21
 - INT 13H—Disk Routines, 8-22
 - Fixed Disk, 8-24
 - INT 14H—Communication Routines, 8-27
 - Get Communications Port Status, 8-30
 - Initialize Communication Port, 8-27
 - Receive Character, 8-29
 - Send Character, 8-29
 - INT 15H—Protected Mode Memory Size Routine, 8-31
 - INT 16H—Keyboard Routines, 8-32
 - Check Keystroke, 8-32
 - Get Current Shift Status, 8-33
 - Read Next Character, 8-32
 - INT 17H—Printer Routines, 8-43
 - Get Printer Port Status, 8-44
 - Initialize Printer Port, 8-44
 - Print a Character, 8-43
 - INT 19H—Bootstrap Routine, 8-45
 - INT 1AH—Time-Of-Day Routine, 8-46
 - INT 20H—Program Terminate, 7-17
 - INT 21H—Function Request, 7-19
 - INT 22H—Terminate Address, 7-20
 - INT 23H—Control-C Exit Address, 7-21
 - INT 24H—Fatal Error Abort Address, 7-22
 - Error Codes, 7-22
 - INT 25H—Absolute Disk Read, 7-27
 - INT 26H—Absolute Disk Write, 7-30
 - INT 27H—Terminate but Stay Resident, 7-33
 - INT 5H—Print Screen Routine, 8-7
 - Intel 8251 UART Pin Assignments, 9-54
 - Interactive Entry, 2-9
 - Interrupt 10H Functions, 10-13
 - Blinking Color Effects for DEB Palettes 0-3, 10-26
 - Comments, 10-22
 - Customized LUT Input, 10-23
 - DEB Settings, 10-15
 - Default Palettes, 10-24
 - Dither Combinations for DEB Palettes 0-3, 10-27
 - Example, 10-15
 - Read Character and Attribute at Cursor Position, 10-19
 - Read Current Video State, 10-29
 - Read Cursor Position, 10-16
 - Read Dot, 10-28
 - Scroll Active Page Down, 10-18
 - Scroll Active Page Up, 10-17
 - Select Active Display Page, 10-16
 - Set Color Palette, 10-21
 - Set Cursor Position, 10-15
 - Set Display Mode, 10-14
 - VDC Settings, 10-15
 - Write Character Only at Cursor Position-20,
 - Write Character and Attribute at Cursor Position, 10-20
 - Write Dot, 10-28
-

Interrupt 10H Functions (Contd)

Write Teletype to Active Page, 10-29

Interrupt 24, A-19

Interrupt Devices, 9-12

Interrupt Priority, 9-44

Interrupting NMI Handlers, A-8

Interrupts, 9-10, 9-50, 9-117

Interrupts (INT), 7-16

Invoking LINK, 2-9

Interactive Entry, 2-9

LINK Prompts, 2-10

Libraries To Be Searched, 2-12

Listing, 2-11

Load Module, 2-11

Object Modules to be Included, 2-11

Ways to Invoke Link, 2-9

I/O, 1-11

I/O Instructions, 9-9

I/O Ports, 9-9

IOCTL Block, 7-170

IOCTL Character, 7-168

IOCTL Data, 7-165

IOCTL Function Call (44H), A-18

IOCTL Retry, 7-180

IOCTL Status, 7-172

IOCTL is Changeable, 7-174

IOCTL is Redirected Block, 7-176

IOCTL is Redirected Handle, 7-178

IOPL, 4-12

IP Value, 6-11

J

JOIN Command, A-12, A-13

JUMP to Boot Code, 5-25

K

K, 1-11

Keep Process, 7-129

Keyboard Commands/Responses, 9-142

Keyboard Interface, 9-139

Example, 9-142

Functions, 9-140

Keyboard

Commands/Responses, 9-142

Modify Status LED Command, 9-143

Registers, 9-140

Sequencing and Timing, 9-142

Set Repeat Rate/Delay

Command, 9-143

L

L (LOAD), 3-32

LABEL Command, A-13

Last Change Date, 5-12, 5-21

Last Change Time, 5-20

Lib-list, 2-17

Libraries To Be Searched, 2-12

Light Pen Emulation Mode Off, 11-15

Light Pen Emulation Mode On, 11-15

/LINENUMBERS, 2-14

LINK Error Messages, 2-22

LINK File Usage, 2-4

Changing Diskettes, 2-6

Ext, 2-5

Filename, 2-4

Pathname, 2-4

Syntax, 2-4

VM.TMP File, 2-5

d., 2-4

-
- LINK Program, 2-1
 - Command Line Entry, 2-17
 - Invoking LINK, 2-9
 - LINK Error Messages, 2-22
 - LINK File Usage, 2-4
 - LINK Switches, 2-13
 - Overview, 2-3
 - Sample Link Session, 2-20
 - Segments, Groups, and Classes, 2-7
 - LINK Prompts, 2-10
 - /DSALLOCATE, 2-13
 - /HIGH, 2-14
 - /LINENUMBERS, 2-14
 - /MAP, 2-14
 - /NO, 2-16
 - /PAUSE, 2-15
 - /STACK: <number>, 2-15
 - LINK Switches, 2-13
 - List of Mouse Functions, 11-7
 - Listfile, 2-17
 - Listing, 2-11
 - Load Module, 2-11
 - Load Overlay, 7-198
 - Load and Execute a Program, 7-194
 - Lock, 7-226
 - Lock Characteristics, A-8
 - Lock Function Call (5C00H), A-13
 - Logical Record Size, 5-12
 - Look-Up Table (LUT), 10-6
 - Low Memory Map, 5-6
 - LSB, MSB, 1-11
 - LUT Addressing, 10-56
- M**
- M, 1-12
 - M (MOVE), 3-35
 - Make Assign List Entry, 7-239
 - Make Assign List Entry Function (5F003H), A-14
 - /MAP, 2-14
 - Media Check, A-16
 - Media Descriptor, 5-26, 5-27
 - Memory Addressing, 4-3, 4-14
 - Aligned and Non-Aligned Words, 4-7
 - Example, 4-4, 4-6
 - Pages, 4-5
 - Protected-Virtual-Address Mode, 4-3
 - Real-Address Mode, 4-3
 - SEGMENTED ADDRESSING, 4-4
 - Memory Configuration, 5-4
 - Memory Map, 5-5
 - Memory and Disk Allocation, 5-1
 - ASCII Strings, 5-8
 - Disk Boot Sector, 5-24
 - Disk Directory, 5-17
 - Disk Formats, 5-33
 - Disk Layout, 5-16
 - File Allocation Table (FAT), 5-27
 - File Control Blocks, 5-10
 - Handles, 5-9
 - Low Memory Map, 5-6
 - Memory Configuration, 5-4
 - Memory Map, 5-5
 - Overview, 5-3
 - ROM BIOS Data Area, 5-7
 - Miscellaneous Enhancements, A-20
 - Mode Setting, 10-10
 - Modes, 9-74
 - Modes of Operation, 9-145
 - Modify Status LED Command, 9-143
 - Monochrome Graphics, 8-18
 - Monochrome Text Mode, 8-16
 - Mouse, 11-1, 11-3
 - Cursor Descriptions—Text and Graphics Cursors, 11-4
 - List of Mouse Functions, 11-7
 - Mouse Function Calls, 11-8
-

Mouse (Contd)

- Notice to System Programmers, 11-5
- Programming Interface from the GWBASIC Interpreter, 11-3

Mouse (Optional), 9-144

- Modes of Operation, 9-145
- Mouse Operation, 9-144
- Scan Codes, 9-144

Mouse Function Calls, 11-8

- Acceleration type, 11-17
- Conditional Hide Cursor, 11-16
- Configuration Type, 11-21
- Define Event Handler, 11-14
- Define Graphics Cursor Block, 11-12
- Define Text Cursor, 11-12
- Get Acceleration, 11-16
- Get Button Press Information, 11-10
- Get Button Release Information, 11-10
- Get Keyboard Emulation Configuration, 11-19
- Get Mouse Mode, 11-18
- Get Mouse Position and Button Status, 11-9
- Hide Cursor, 11-9
- Light Pen Emulation Mode Off, 11-15
- Light Pen Emulation Mode On, 11-15
- Mouse Initializations, 11-8
- Parameter Structure, 11-19, 11-21
- Read Mouse Motion Counters, 11-13
- Set Acceleration, 11-17
- Set Grid, 11-18
- Set Keyboard Emulation Configuration, 11-20
- Set Minimum and Maximum X Position, 11-11
- Set Minimum and Maximum Y Position, 11-11
- Set Mouse Cursor Position, 11-9
- Set Mouse Mode, 11-18

Mouse Function Calls (Contd)

- Set Mouse Motion/Pixel Ratio, 11-15
- Show Cursor, 11-8

Mouse Initializations, 11-8

Mouse Operation, 9-144

Move File Pointer, 7-160

MS-DOS 3.1 Versus MS-DOS 2.X, A-12

- Argv[0] environment, A-15
- Attribute Field, A-16
- Block Device Drivers, A-16
- Bug Fixes from MS-DOS 2.11, A-21
- COMMAND.COM, A-17, A-20
- CONFIG.SYS, A-17
- Cancel Assign List Entry Function (5F004H), A-14
- Command Codes, A-16
- Create New File Function (5BH), A-15
- Create Temporary File Function (5AH), A-15
- Enhancements to Function Calls and Interrupts, A-18
- File Allocation Table, A-17
- General Enhancements, A-15
- Get Country Data Function (38H), A-19
- Get Extended Error Function (59H), A-15
- Get Machine Name Function (5E00H), A-14
- Get Program Segment Prefix Address Function (62H), A-15
- Get/Set Allocation Strategy Function (58H), A-14
- IOCTL Function Call (44H), A-18
- Installable Device Drivers, A-16
- Interrupt 24, A-19
- JOIN Command, A-12, A-13
- LABEL Command, A-13
- Lock Function Call (5C00H), A-13
- MS-LINKER, A-20

MS-DOS 3.1 Versus MS-DOS 2.X

(Contd)

- Make Assign List Entry Function (5F003H), A-14
- Media Check, A-16
- Miscellaneous Enhancements, A-20
- New Function Call Parameters, A-16
- New General Commands, A-13
- New Network Access Calls, A-13
- New Network Commands, A-12
- New Network Function Calls, A-14
- New Public Function Calls, A-14
- Open Function Call (3DH)-File Access Modes, A-18
- PRINT.COM, A-17
- PRINTF Message Files, A-20
- Printer Setup Function (5E02H), A-14
- Proceed Command, A-20
- SHARE Command, A-12
- SUBST Command, A-12, A-13
- Status Word, A-16
- The Country Variable, A-20
- Unlock Function Call (5C01H), A-13

MS-DOS 3.10 Operating System

Documentation, 1-20

MS-LINKER, A-20

N

- N (NAME), 3-37
- N/A, 1-12
- Name, 1-9
- National 8250 UART Pin Assignments, 9-53
- New Function Call Parameters, A-16
- New General Commands, A-13

- New Network Access Calls, A-13
- New Network Commands, A-12
- New Network Function Calls, A-14
- New Public Function Calls, A-14
- Nibble, 1-9
- /NO, 2-16
- Notice to System Programmers, 11-5
- NPX, 1-12
- NT, 4-12
- Number of FAT Sectors, 5-26
- Number of FATs, 5-25
- Number of Heads, 5-26
- Number of Hidden Sectors, 5-26
- Number of Root Directory Entries, 5-25
- Number of Sectors in Logical Image, 5-25
- Numeric Exceptions, A-6
- Numerical Table of Functions, 7-9

O

- O (OUTPUT), 3-40
- Object Modules to be Included, 2-11
- Obj-list, 2-17
- OEM Name and Version, 5-25
- OF, 4-11
- Open File, 7-64
- Open Function Call (3DH)-File Access Modes, A-18
- Open Handle, 7-150
- Other Uses of the PSP, 6-18
- Overlay Modes, 10-6, 10-11
- Overlay Modes—LUT
- Programming, 10-46

P

P (PROCEED), 3-41

Pages, 4-5

Parallel Printer Interface, 9-147

Example, 9-150

Functions, 9-148

Registers, 9-148

Sequencing and Timing, 9-149

Parameter Area, 6-14

Parameter Block Contents, 7-196, 7-199

Parameter Structure, 11-19, 11-21

Parameters, 9-122

Parse File Name, 7-113

Parts Catalog, 1-17

Pathname, 1-9, 2-4

/PAUSE, 2-15

PF, 4-11

Pointer to Device Interrupt Entry

Point, 9-19

Pointer to Device Strategy Entry

Point, 9-19

Pointer to Next Device, 9-15

Pre-Defined Handles, 5-9

Print Character, 7-44

Print a Character, 8-43

PRINT.COM, A-17

Printer Setup, 7-234

Printer Setup Function (5E02H), A-14

PRINTF Message Files, A-20

Proceed Command, A-20

Program Loading Process, 6-15

Initial Conditions, 6-17

Other Uses of the PSP, 6-18

PSP Conditions Upon Program

Initiation, 6-15

of Bytes, 6-13

Program Segment Prefix, 6-12

Allocation Block, 6-13

Area 1, 6-14

Area 2, 6-14

Control-C Exit Address, 6-13

Error Exit Address, 6-13

Format, 6-12

Function Dispatch Call, 6-13

INT 20H, 6-13

Parameter Area, 6-14

Reserved for MS-DOS, 6-13

Terminate Address, 6-13

Program Structure and Loading, 6-1

EXE2BIN Program, 6-6

File Header Format, 6-9

Overview, 6-2

Program Loading Process, 6-15

Program Segment Prefix, 6-12

Pros And Cons For Selecting a

Program File Format, 6-3

Relocation Process For .EXE

Files, 6-20

Programmable Devices, 9-8

Programmable Interrupt Controller
(PIC), 9-151

Example, 9-160

Registers, 9-152

Sequencing and Timing, 9-159

Programmable Interval Timer, 9-161

Example, 9-165

Functions, 9-163

Registers, 9-163

Sequencing and Timing, 9-164

Programming Considerations, 7-15

Calling From Macro Assembler, 7-15

Calling from a High-Level

Language, 7-15

GW BASIC, 7-15

Programming Devices Directly, 1-14

Programming Interface from the
 GWBASIC Interpreter, 11-3
 Programming Steps, 1-13
 Programming Devices Directly, 1-14
 Running High-Level Language, 1-13
 Writing Assembler Programs, 1-13
 Writing Utilities, 1-13
 Programming Tips, 10-8
 Hardware/Software
 Compatibility, 10-8
 Setup, 10-9
 Programming the Bit Planes, 10-56
 Programming the LUT, 10-30
 16-Color Graphics LUT
 Programming, 10-31
 LUT Addressing, 10-56
 Overlay Modes—LUT
 Programming, 10-46
 Programming the Bit Planes, 10-56
 Short LUT Addresses, 10-58
 Timing Bits, 10-57
 01-6,
 Pros And Cons For Selecting a
 Program File Format, 6-3
 Cons for .COM, 6-4
 Cons for .EXE, 6-3
 Pros for .COM, 6-4
 Pros for .EXE, 6-3
 Pros for .COM, 6-4
 Pros for .EXE, 6-3
 Protected-Virtual-Address Mode, 4-3
 PSP, 1-12
 PSP Conditions Upon Program
 Initiation, 6-15
 Purpose of This Guide, 1-3

Q

Q (QUIT), 3-42
 Quotients of 80H or 8000H, A-8

R

R (REGISTER), 3-44
 RAM, 1-12
 RAM Diagnostic Function, 9-132
 Random Block Read, 7-107
 Random Block Write, 7-110
 Random Read, 7-93
 Random Write, 7-96
 Read Attribute or Character, 8-12
 Read Character and Attribute at
 Cursor Position, 10-19
 Read Current Video State, 8-15, 10-29
 Read Cursor Position, 8-10, 10-16
 Read Dot, 10-28
 Read ECC Burst Error Length
 Function, 9-130
 Read Function, 9-128
 Read Handle, 7-154
 Read Keyboard, 7-51
 Read Keyboard and Echo, 7-38
 Read Light Pen Position, 8-10
 Read Long Function, 9-133
 Read Mouse Motion Counters, 11-13
 Read Next Character, 8-32
 Read Pixel, 8-15
 Read Sector Buffer Function, 9-131
 Read Verify Function, 9-126
 Real-Address Mode, 4-3
 Real-Time Clock And Calendar, 9-166
 Example, 9-171
 Functions, 9-167
 Registers, 9-167

Real-Time Clock And Calendar (Contd)

- Sequencing and Timing, 9-171
- Recalibrate Function, 9-124
- Receive Character, 8-29
- Record Length, 9-20
- Register Formats, 9-173
- Register Indirect Addressing, 4-16
- Register Operand Addressing, 4-14
- Register and Immediate Addressing, 4-14
- Registers, 4-8
 - General Registers, 4-8
 - Segment Registers, 4-9
 - Status and Control Registers, 4-9
- Relative Offset, 6-11
- Relative Record Number, 5-14
- Relocation Process For .EXE Files, 6-20
- Remove Directory, 7-144
- Rename File, 7-84
- Request Header, 9-20
 - Command Code, 9-21
 - Format, 9-20
 - Record Length, 9-20
 - Reserved, 9-34
 - Status, 9-33
 - Unit Code, 9-20
- Request Sense Function, 9-124
- Reserved Sectors, 5-25
- Reserved for MS-DOS, 6-13
- Reset Disk, 7-61
- ROM, 1-12
- ROM BIOS Data Area, 5-7
- ROM BIOS Listing, 8-48
- Routine Interrupt Table, 8-6
- Routines, 8-5
- Runfile, 2-17
- Running High-Level Language, 1-13

S

- S (SEARCH), 3-48
- Sample Link Session, 2-20
- Scan Codes, 9-144
- Scroll Active Page Down, 8-12, 10-18
- Scroll Active Page Up, 8-11, 10-17
- Search for First Entry, 7-69
- Search for Next Entry, 7-72
- Sectors per Allocation Unit, 5-25
- Sectors per Track, 5-26
- Seek Function, 9-129
- Segment, 2-7
- Segment Registers, 4-9
- Segmented Addressing, 4-4
- Segments, Groups, and Classes, 2-7
 - Class, 2-8
 - Group, 2-7
 - Segment, 2-7
- Select Active Display Page, 10-16
- Select Active Page Number, 8-11
- Select Disk, 7-63
- Send Character, 8-29
- Sequential Read, 7-76
- Sequential Write, 7-79
- Service Information, 1-15
- Set Acceleration, 11-17
- Set Block, 7-192
- Set Color Palette, 8-14, 10-21
- Set Cursor Position, 8-10, 10-15
- Set Cursor Type, 8-9
- Set Date, 7-119
- Set Disk Transfer Address, 7-87
- Set Display Mode, 10-14
- Set Grid, 11-18
- Set Interrupt Vector, 7-104
- Set Keyboard Emulation Configuration, 11-20

-
- Set Minimum and Maximum X Position, 11-11
 - Set Minimum and Maximum Y Position, 11-11
 - Set Mode and Clear Screen, 8-9
 - Set Mouse Cursor Position, 11-9
 - Set Mouse Mode, 11-18
 - Set Mouse Motion/Pixel Ratio, 11-15
 - Set Relative Record, 7-102
 - Set Repeat Rate/Delay Command, 9-143
 - Set Time, 7-123
 - Set/Reset Verify Flag, 7-125
 - Setting Colors and Effects, 10-12
 - Setting the Baud Rate, 9-42
 - SF, 4-11
 - SHARE Command, A-12
 - Short LUT Addresses, 10-58
 - Show Cursor, 11-8
 - Size of File, 6-10
 - Size of Header, 6-10
 - Software Compatibility Considerations, A-4
 - Duplicate Prefixes, A-7
 - Exception Handlers Prefixes, A-6
 - Exception Points to the DIV Instruction, A-6
 - External Interrupt Handlers, A-8
 - Far JMP Instruction at FFFF0H, A-6
 - Instruction Clock Cycles, A-5
 - Interrupting NMI Handlers, A-8
 - Lock Characteristics, A-8
 - Numeric Exceptions, A-6
 - Quotients of 80H or 8000H, A-8
 - Shifting More Than 31 Bits, A-7
 - Six Additional Interrupt Vectors A-4,
 - Undefined Operations, A-6
 - Value Written by Push SP, A-7
 - Source String, 7-240
 - SP Value, 6-11
 - Speaker, 9-172
 - Example, 9-174
 - Registers, 9-173
 - Resister Formats, 9-173
 - Sequencing and Timing, 9-174
 - /STACK: <number>, 2-15
 - Stack Value, 6-10
 - Standard FCB, 5-15
 - Standard FCB Format, 5-10
 - Status, 9-33
 - Status Flags, 4-10
 - Status Word, A-16
 - Status and Control Registers, 4-9
 - SUBST Command, A-12, A-13
 - Sum of Words, 6-11
 - Switching between Real and Protected Modes, 9-175
 - System Calls, 7-1
 - Absolute Disk Read, 7-27
 - Absolute Disk Write, 7-30
 - Alphabetical Table of Functions, 7-12
 - Control-C Exit Address, 7-21
 - Fatal Error Abort Address, 7-22
 - Function Request, 7-19
 - Functions, 7-35
 - General Macros, 7-245
 - Interrupts (INT), 7-16
 - Numerical Table of Functions, 7-9
 - Overview, 7-7
 - Program Terminate, 7-17
 - Programming Considerations, 7-15
 - Table of Interrupts, 7-8
 - Terminate Address, 7-20
 - Terminate but Stay Resident, 7-33
 - System Programming Concepts, 1-1
 - Abbreviations Used in This Guide, 1-10
 - Conventions Used in This Guide, 1-4
-

System Programming Concepts (Contd)
 General Documentation—AT&T
 Personal Computer 6300 PLUS, 1-18
 MS-DOS 3.10 Operating System
 Documentation, 1-20
 Parts Catalog, 1-17
 Programming Steps, 1-13
 Purpose of This Guide, 1-3
 Service Information, 1-15
 Technical Documentation—AT&T
 Personal Computer 6300 PLUS, 1-19
 Terms Used in This Guide, 1-8
 UNIX System Software Development
 Documentation, 1-23
 UNIX System V Release 2.0
 Foundation Set Documentation, 1-21

T

T (TRACE), 3-50
 Table of Interrupts, 7-8
 Technical Documentation—AT&T
 Personal Computer 6300 PLUS, 1-19
 Terminate Address, 6-13
 Terminate Program, 7-36
 Terms Used in DEBUG, 3-4
 Address, 3-4
 Drive, 3-5
 Keyletter, 3-5
 Parameter, 3-5
 Range, 3-5
 Value, 3-6
 Terms Used in This Guide, 1-8
 Bit, 1-8
 Byte, 1-8
 Chapter, 1-8
 Double Word, 1-9
 Extension, 1-9
 Filename, 1-9

Terms Used in This Guide (Contd)
 Name, 1-9
 Nibble, 1-9
 Pathname, 1-9
 Word, 1-9
 Test Drive Ready Function, 9-123
 Text Mode, 9-74
 TF, 4-12
 Timing Bits, 10-57
 Transparent Mode, 10-11
 01-5,
 Type Styles, 1-5

U

U (UNASSEMBLE), 3-52
 UART, 1-12
 Undefined Operations, A-6
 Unit Code, 9-20
 UNIX System Software Development
 Documentation, 1-23
 UNIX System V Release 2.0 Foundation
 Set Documentation, 1-21
 Unlock, 7-229
 Unlock Function Call (5C01H), A-13

V

Value Written by Push SP, A-7
 Variable Addressing, 9-10
 VDC Settings, 10-15
 VM.TMP File, 2-5

W

W (WRITE), 3-54
Ways to Invoke Link, 2-9
Word, 1-9
Write Attribute or Character, 8-13
Write Character, 8-13
Write Character Only at Cursor
Position-20
Write Character and Attribute at
Cursor Position, 10-20
Write Dot, 10-28
Write Function, 9-128
Write Handle, 7-156
Write Long Function, 9-134
Write Pixel, 8-14
Write Sector Buffer Function, 9-131
Write Teletype, 8-15
Write Teletype to Active Page, 10-29
Writing Assembler Programs, 1-13
Writing Utilities, 1-13

Z

ZF, 4-11

/

/DSALLOCATE, 2-15
/HIGH, 2-16
/LINENUMBERS, 2-16
/MAP, 2-16
/NO, 2-18
/PAUSE, 2-17
/STACK: <number>, 2-18